



apricot



apricot



apricot



apricot



apricot

# USING THE GW-BASIC INTERPRETER



# The GW-BASIC 3.1 Interpreter

1

Microsoft wrote the first BASIC interpreter for microcomputers in 1975, and Microsoft BASIC now has over 1,000,000 installations. Users, manufacturers and software vendors have written application programs in Microsoft BASIC, and it is found on all successful microcomputers.

BASIC is a general purpose programming language that can be used for business, science, games and education. It is interactive, so without writing a program for the purpose you can carry out processes, calculations and test existing programs.

Microsoft GW-BASIC<sup>®</sup> is the most extensive implementation of Microsoft BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many features rarely found in other BASIC interpreters. In addition, the Microsoft GW-BASIC Interpreter has sophisticated screen handling, graphics, and structured programming features that are especially suited for application development.

GW-BASIC includes several features not found in other BASICs, and takes advantage of the MS-DOS<sup>®</sup> environment to enhance programming power.

These features include:

- \* Facilities for writing programs within a network environment
- \* Re-direction of Standard Input (INPUT, LINE INPUT) and Standard Output (PRINT)
- \* Improved Disk I/O facilities for handling larger files
- \* Multi level directories for better disk organisation
- \* Directory management (MD/CD/RD)
- \* Improved Graphics: Line Clipping, VIEW, WINDOW
- \* Screen Editor enhancements including text window support
- \* Additional Event Trapping: PLAY, TIMER

- \* User definable Keyboard trapping
- \* Double Precision Transcendentals (optional with the /D switch)
- \* More precise control of BASIC's memory allocation for user routines with the /M: switch

# Starting the GW-BASIC Interpreter

## 1.1

To start the GW-BASIC Interpreter, enter,

```
GW BASIC
```

To run a particular program as soon as GW-BASIC starts, enter:

```
GW BASIC <filespec>
```

<filespec> is the filename for the program. It is preceded by an optional device designator, and followed by an optional extension name. For example, to start the program FILE.BAS which is on disk drive A:, enter

```
GW BASIC A:FILE.BAS
```

To leave GW-BASIC and return to MS-DOS, enter:

```
SYSTEM
```

## 1.2 Command Line Option Switches

Option switches alter the way the Interpreter works. To use an option, enter GW-BASIC followed by the option you require:

GW-BASIC    [ <stdin ]  
              [ >stdout ]  
              [ <filespec> ]  
              [ /D ]  
              [ /F: < number of files> ]  
              [ /I ]  
              [ /M: [ < highest memory location> ]  
              [ < maximum block size> ]  
              [ /S: < lrecl> ]

### <stdin

For example <SIMON. This redirects BASIC input from the file SIMON. If used, <stdin must appear before any other switches. You must enter the less-than (<) character with stdin - it is not part of the command format conventions used in this manual.

### >stdout

For example >HANNAH. This redirects BASIC output to the file specified by >stdout. If used, >stdout must appear before any other switches, and after <stdin if used. To prevent an existing file being overwritten, enter two greater than characters (>>). This appends output to the file rather than overwrites it. You must enter the greater than character(s) (>) with stdout — they are not part of the command format conventions used in this manual.

### **< filespec >**

The file specification of a BASIC program.

If used, the effect is as if a RUN <filespec> command is given straight after starting BASIC.

To start BASIC programs from a batch file, put this command option in an AUTOEXEC.BAT file. Programs run in this way must exit via the SYSTEM statement so that the next command from the AUTOEXEC.BAT file can be executed.

### **/F: < number of files >**

This switch is ignored unless the /I switch is also specified - see /I below. /F sets the maximum number of files which may be open simultaneously during the execution of a BASIC program. Use decimal, octal (precede by &O), or hexadecimal (precede by &H). Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. (The data buffer size may be altered via the /S: option switch.) If /F: is omitted, the number of files is set to three.

The number of open files that MS-DOS supports depends upon the value of the FILES= parameter in the CONFIG.SYS file. We recommend that FILES= 10 for BASIC. The first three are required for system use, plus one for loading, leaving six for BASIC file I/O. So /F:6 is the maximum supported by MS-DOS when FILES= 10 appears in the CONFIG.SYS file. Attempts to OPEN a file after all the file handles have been exhausted result in the error "Too many files".

### **/S: < lrecl >**

This switch is ignored unless the /I switch is also specified - see /I below. <lrecl> sets the maximum record size allowed for use with random files. Use decimal, octal (precede by &O), or hexadecimal (precede by &H). The record size option to the OPEN statement cannot exceed the value set here. If omitted, the record size defaults to 128 bytes.

### **/D**

This switch ensures the Double Precision Transcendental maths package remains resident. If omitted, the package is deleted and the space freed for program use.

/I

GW-BASIC can dynamically allocate space for file operations, and does not need the /S and /F switches. However, for some existing applications some BASIC internal data structures must be static. To provide compatibility with these BASIC programs, GW-BASIC statically allocates space required for file operations based on the /S and /F switches when the /I switch is used.

**/M:[ <highest memory location> ]  
[, <max block size> ]**

This switch sets the highest memory location to be used by BASIC. Set it in decimal, octal (precede by &O), or hexadecimal (precede by &H). BASIC will attempt to allocate 64k of memory for the data and stack segment. Use /M: to set the highest location that BASIC can use if machine language subroutines are to be used with BASIC programs. When omitted or set to 0, BASIC attempts to allocate all it can up to a maximum of 65536 bytes.

To load items above the highest set location, use the optional parameter <maximum block size> to preserve the necessary space. Set it in paragraphs (byte multiples of 16); it can be in decimal, octal (precede by &O), or hexadecimal (precede by &H). When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes (65536 = 4096 x 16) for BASIC's data and stack segment. For example, for 65536 bytes for BASIC and 512 bytes for machine language subroutines, use /M:,&H1010 (4096 paragraphs for BASIC + 16 for the routines).

The maximum block size option can be used to shrink the BASIC block to free more memory.

/M:,2048 allocates 32768 bytes for data and stack.

/M:32000,2048 allocates 32768 bytes maximum for you to use, but BASIC will only use the lower 32000.

This leaves 768 bytes.

Examples of command line options

A>GW-BASIC PAYROLL

Using 64k of memory and 3 files, load and execute PAYROLL.BAS.

A>GW-BASIC INVENT/I/F:6

Using 64k of memory and 6 files, load and execute INVENT.BAS.

A>GW-BASIC /I/F:4/S:512

Use 4 files and allow a maximum record length of 512 bytes.

## 1.3 Modes of Operation

The GW-BASIC Interpreter works in two modes: direct or indirect.

In direct mode, statements and commands are carried out as you enter them, without line numbers. After each direct statement and `RETURN` the screen displays Ok. Results of arithmetic and logical operations are shown immediately and can be stored, but the instructions themselves are lost.

Direct mode is useful for debugging programs and for using the GW-BASIC Interpreter as a calculator.

Use indirect mode for entering programs. Program lines here start with line numbers, and you can subsequently store them to be run at any time with the RUN command.

# Line format

## 1.4

GW-BASIC program lines use the format:

< nnnnn > < BASIC statement > [:BASIC statement...]  
RETURN

More than one statement can be on a line, but must be separated from the previous one by a colon (:).

Program lines always begin with a line number — in the range zero to 65529 — and end with RETURN.

Line numbers give the order in which program lines are stored in memory, and are also used as references in branching and editing. Each line may have up to 255 characters. At the end of each physical line the text “wraps” automatically, keeping the logical line intact.

Use a full stop (.) in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

## 1.5 Active and Visual (Display) Pages

Set the size of these pages with SCREEN statement.  
See Section 5 for more details.

# LEARNING THE LANGUAGE



## Learning the language and using advanced features

2

This section introduces the BASIC character set, language rules for constants, variables and expressions and also includes information on using advanced features

## ***Control Characters***

GW-BASIC uses the following control characters:

<b>Control Character</b>	<b>Action</b>
Control-A	Enters edit mode on the line being typed.
Control-B	Moves cursor to previous word.
Control-C	With the interpreter, interrupts program execution and returns to BASIC command level.
Control-E	Clears to end of line.
Control-F	Moves cursor to the next word.
Control-G	Sounds the speaker.
Control-H	Backspaces. Deletes the last character typed.
Control-I	Tabs to the next tab stop. Tab stops are set every eight columns
Control-K	Sends cursor to home location.
Control-L	Clears the screen.
Control-N	Moves cursor to the end of the line.
Control-O	Halts program output while execution continues. A second Control-O resumes output.
Control-Q	Resumes program execution after a Control-S.
Control-R	Toggles the insert and typeover modes.
Control-S	Suspends program execution.
Control-T	Updates the function key display line.
Control-U	Deletes the line currently being typed.
Control-W	Deletes the word currently at the cursor.
Control-X	Displays the next program line if the line at the cursor starts with a number.
Control-Y	Displays the previous program line if the line at the cursor starts with a number.
Control-Z	Clears from the cursor to the end of the screen

### ***Commonly Used Commands***

GW-BASIC provides you with the facility to action some of its more commonly used commands by holding down the ALT key and pressing another key. The list below shows the commands which can be actioned by this method.

<b>Keystroke</b>	<b>Command</b>
ALT-A	AUTO
ALT-B	BSAVE
ALT-C	COLOR
ALT-D	DELETE
ALT-E	EDIT
ALT-F	FOR
ALT-G	GOTO
ALT-H	HEX\$
ALT-I	INPUT
ALT-J	J
ALT-K	KEY
ALT-L	LOCATE
ALT-M	MID\$
ALT-N	NEXT
ALT-O	OPEN
ALT-P	PRINT
ALT-Q	Q
ALT-R	RUN
ALT-S	SCREEN
ALT-T	THEN
ALT-U	USING
ALT-V	VAL
ALT-W	WIDTH
ALT-X	XOR
ALT-Y	Y
ALT-Z	Z

Constants are values which cannot be changed during execution. BASIC provides two types: string and numeric.

### ***String Constants***

A string constant is a sequence of up to 255 letters and control characters enclosed in double quotation marks:

```
"HELLO"  
"$25,000.00"  
"Number of Employees"
```

### ***Numeric constants***

A numeric constant is a positive or negative number. They cannot contain commas. There are five types of numeric constant as follows:

#### ***Integer constants***

Whole numbers between  $-32768$  and  $32767$ , without decimal points.

#### ***Fixed-point constants***

Positive or negative real numbers - that is, numbers with decimal points.

#### ***Floating-point constants***

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent).

The allowable range for floating-point constants is  $10^{-38}$  to  $10^{+38}$ .

## Examples

```
235.988E-7
```

equals

```
.0000235988
```

```
2359E6
```

equals

```
2359000000
```

Double precision floating-point constants are denoted by the letter D instead of E.

## *Hex constants*

These are hexadecimal numbers, denoted by the prefix &H. Hex constants cannot be greater than decimal 64K.

## Example

```
&H76  
&H32F
```

## *Octal constants*

Octal numbers, denoted by the prefix &O or &. They cannot exceed decimal 64K.

## Example

```
&O347  
&1234
```

## ***Single/Double Precision Numeric Constants***

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with seven digits of precision and printed with up to six digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant with :

- \* Seven or fewer digits, or
- \* Exponential form using E, or
- \* A trailing exclamation mark (!)

### **Example**

```
46.8  
-1.09E-06  
3489.0  
22.5!
```

A double precision constant is any numeric constant with:

- \* Eight or more digits, or
- \* Exponential form using D, or
- \* A trailing number sign (#)

### **Example**

```
345692811  
-1.09432D-06  
3489.0#  
7654321.1234
```

Variables are names representing values used in a GW-BASIC program. They can represent a numeric value or a string. You can assign values to a variable explicitly, or they may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero (for a numeric variable) or null (for a string).

### ***Variable Names and Declaration Characters***

Variable names can be any length, but only the first 40 characters are significant. The first character must be a letter, but the rest can be letters, numbers, and the decimal point. Special type declaration characters (listed below) are also allowed.

A variable name cannot be a reserved word. These include GW-BASIC commands, statements, function names, and operator names. Embedded reserved words are allowed, but no variable may start with the letters `USR`. A variable beginning with `FN` is assumed to be a call to a user-defined function.

String variable names are written with a dollar sign (\$) as the last character. For example:

```
A$ = "SALES REPORT".
```

The dollar sign is a variable type declaration character; that is, it *declares* that the variable will represent a string.

Numeric variable names can be declared as integer, single precision, or double precision values. The type declaration characters for these are:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default is single precision. If a number specified in a program has too many significant digits to be represented as a single precision number, it is represented as a double precision number, and the “#” which signifies double precision follows the number in the program listing.

Integer variables produce the fastest and most compact object code. For example, the following program performs approximately 30 times faster when the loop control variable “I” is replaced with “I%”, or when I is declared an integer variable with DEFINT.

```
100 FOR I=1 TO 10
120 A(I)=0
140 NEXT I
```

### Example variable names:

PI# Declares a double precision value.

MINIMUM! Declares a single precision value.

LIMIT% Declares an integer value.

N < Declares a string value.

ABC Represents a single precision value.

The default variable type may be selectively changed by using statements DEFINT/SNG/DBL/ STR, - see these headings in Section 5.

## ***Array Variables***

An array is a group or table of values held under the same variable name. Each element in an array is referenced by an array variable subscripted by an integer or integer expression. For example,  $V(10)$  references element number ten in the array  $V$ . An array can have more than one dimension, so that  $T(1,4)$  references a value in the two-dimensional array  $T$ . The array variable name has as many subscripts as the array has dimensions —  $G(2,56,4,34)$  in a four-dimensional array, for example. Arrays may have up to 255 dimensions, with up to 32,767 elements per dimension. The maximum amount of space an array can occupy is 64K.

## ***Space Requirements***

The following list gives only the number of bytes occupied by the values represented by the variable names. Additional requirements may vary according to implementation.

	Type	Bytes
Variables	Integer	2
	Single precision	4
	Double precision	8
	String	3
Arrays	Integer	2 per element
	Single precision	4 per element
	Double precision	8 per element
	String	3 per element

# Expressions and Operators

## 2.4

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators. An expression always produces a single value.

Operators perform mathematical or logical operations on values. They are in three categories - Arithmetic, Relational and Logical

### ***Precedence of Operations***

Operators in a program statement have an order of precedence, as shown below. Where operations are of the same level of precedence, the leftmost is carried out first and the rightmost last.

Exponentiation

Negation

Multiplication & Division

Integer Division

Modulus Arithmetic

Addition & Subtraction

Relational Operators

NOT

AND

OR & XOR

EQV

IMP

## Arithmetic Operators

These, in order of evaluation, are:

Operator	Operation	Sample expression
$\wedge$	Exponentiation	$X^Y$
$-$	Negation	$-X$
$*, /$	Multiplication, Floating-point Division	$X*Y$ $X/Y$
$\backslash$	Integer division	$12 \backslash 6 = 2$
MOD	Modulus arithmetic	$10 \text{ MOD } 4 = 2$ ( $10/4 = 2$ with remainder 2)
$+, -$	Addition, Subtraction	$X + Y$

To change this order use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Sample algebraic expressions and their GW-BASIC counterparts:

Algebraic Expression	BASIC Expression
$X + 2Y$	$X + Y * 2$
$\frac{X - Y}{Z}$	$(X - Y) / Z$
$\frac{XY}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$(X^2)^Y$	$(X^2)^Y$
$X^{(YZ)}$	$X^{(YZ)}$
$X(-Y)$	$X * (-Y)$

Two consecutive operators must be separated by parentheses.

## ***Integer Division and Modulus Arithmetic***

Integer division is shown by the backslash (\).

Operands must be in the range  $-32768$  to  $32767$ .

They are rounded to integers before division, and the quotient is truncated to an integer.

### **Example**

```
100 LET DIV1 = 10\4
200 LET DIV2 = 25.68\6.99
300 PRINT DIV1, DIV2
```

returns

```
2           3
```

Modulus arithmetic is shown by the operator MOD.

It returns the integer that is the remainder of an integer division.

### **Example**

```
10 PRINT 10.4 MOD 4, 25.68 MOD 6.99
```

returns

```
2           5
```

$(10\backslash 4 = 2, \text{ remainder } 2; 26\backslash 7 = 3, \text{ remainder } 5)$

## ***Overflow and Division by Zero***

Division by zero in an expression causes the error message "Division by zero". Machine infinity (the largest number represented in floating-point format) with the sign of the numerator is supplied as the result of the division, and the program continues.

Where the evaluation of an exponentiation operator raises zero to a negative power, "Division by zero" is displayed. Positive machine infinity is supplied as the result of the exponentiation and the program continues.

If overflow occurs, the interpreter displays an 'Overflow' message, supplies machine infinity with the algebraically correct sign as the result and continues.

## ***Relational Operators***

Relational operators compare two values, to return a result of *true* (−1) or *false* (0).

You can then use the result to make a decision on program flow - see the IF statement in Section 5.

The relational operators are

<b>Operator</b>	<b>Relation Tested</b>	<b>Example</b>
=	Equality	X = Y
< >	Inequality	X < > Y
<	Less than	X < Y
>	Greater than	X > Y
< =	Less than or equal to	X < = Y
> =	Greater than or equal to	X > = Y

The equal sign is also used to assign a value to a variable - see the LET statement in Section 5.

When arithmetic and relational operators are combined in one expression, arithmetic is always performed first.

## Example

```
X+Y<(T-1)/Z
```

is true if the value of X plus Y is less than the value of T - 1 divided by Z.

```
IF SIN(X)<0 GOTO 1000  
IF I MOD J<>0 THEN K=K+1
```

## Logical Operators

Logical operators perform bit-by-bit calculation and return a result of *true* (not zero) or *false* (zero). Logical operations are performed after arithmetic and relational operations in an expression. The table below lists the operators in order of precedence and gives the outcome of logical operations.

### GW-BASIC Relational Operators Truth Table

Operation	Value	Value	Result
NOT			
	X		NOT X
	T		F
	F		T
AND			
	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR			
	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F

Operation	Value	Value	Result
XOR			
	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
EQV			
	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T
IMP			
	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T

Logical operators, like relational operators, can connect two or more relations and return a *true* or *false* value to be used in a decision - see IF statements in Section 5.

### Example

```
IF D < 200 AND F < 4 THEN 80
IF I > 10 OR K < 0 THEN 50
IF NOT P THEN 100
```

Logical operators convert their operands to 16-bit, signed, two's complement integers in the range  $-32768$  to  $32767$ . (If the operands are not in this range, an error results.) If both operands are supplied as 0 or  $-1$ , logical operators return 0 or  $-1$ . The given operation is performed on these integers bit-by-bit - that is, each bit of the result is determined by the corresponding bits in the two operands.

Therefore you can use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can be used to *mask* all but one of the bits of a status byte at a machine I/O port.

The OR operator can be used to *merge* two bytes to create a particular binary value. The following examples, all using decimal numbers, show how logical operators work.

10 AND 7 = 2	10 = binary 1010 and 7 = binary 111, so 10 AND 7 = 2.
16 AND 15 = 0	16 = binary 10000 and 15 = binary 1111, so 16 AND 15 = 0.
-1 AND 8 = 8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
10 OR 10 = 10	10 = binary 1010, so 1010 OR 1010 = 1010 (decimal 10).
-1 OR -2 = -1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X = -(X+1)	The two's complement of any integer is the bit complement plus one.

## String Operators

You can concatenate strings by using the plus sign (+).

### Example

```
10 A$="FILE" : B$="NAME"  
20 PRINT A$+B$  
30 PRINT "NEW" +A$+ B$
```

returns

```
FILENAME  
NEW FILENAME
```

The same relational operators used with numbers can compare strings:

= < > < > < = > =

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

### For example

"AA"	is less than	"AB"
"FILENAME"	is equal to	"FILENAME"
"X&"	is greater than (because # comes before &)	"X#"
"CL "	is greater than (because of the trailing space)	"CL"
"kg"	is greater than	"KG"
"SMYTH"	is less than	"SMYTHE"
B\$	is less than (where B\$="8/12/78")	"9/12/78"

Therefore string comparisons can test string values or compare strings alphabetically. All string constants used in comparison expressions must be enclosed in quotation marks.

## Type conversion

2.5

When necessary, GW-BASIC converts a numeric constant from one type to another. The following rules and examples apply to conversions.

If a numeric variable of one type is set equal to a numeric constant of a different type, the number is stored as the type declared in the variable name.

### Example

```
10 PERCENT% = 23.42  
20 PRINT PERCENT%
```

returns

```
23
```

During expression evaluation, all the operands in an arithmetic or relational operation are converted to the same degree of precision as that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

### Examples

```
10 DEDUCTION# = 6#/7  
20 PRINT DEDUCTION#
```

returns

```
.8571428571428571
```

The arithmetic was performed in double precision and the result returned in DEDUCTION# as a double precision value.

```
10 DEDUCTION = 6#/7  
20 PRINT DEDUCTION
```

returns

```
.857143
```

The arithmetic was performed in double precision, and the result rounded to single precision and returned to DEDUCTION (single precision variable) and printed.

Logical operators convert their operands to integers and return an integer result. Operands must be in the range  $-32768$  to  $32767$  or an "Overflow" error occurs.

When a floating-point value is converted to an integer, the fractional portion is rounded.

### Example

```
10 CASH% = 55.88  
20 PRINT CASH%
```

returns

```
56
```

If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than  $6.3E-8$  times the original single precision value.

## Example

```
10 A = 2.04  
20 B# = A  
30 PRINT A;B#
```

returns

```
2.04                2.039999961853027
```

GW-BASIC incorporates two kinds of functions: intrinsic and user-defined.

### **Intrinsic Functions**

When a function is used in an expression it calls a predetermined operation which is to be performed on an operand. GW-BASIC has built-in functional operators, such as SQR (square root) or SIN (sine). These are called *"intrinsic functions"*.

### **User-Defined Functions**

You can make your own functions with GW-BASIC -see the DEF FN Statement in Section 5.

This section gives an overview of event trapping. For details on individual statements see Section 5.

Event trapping can only happen when GW-BASIC is running a program. It allows a program to transfer control to a specific program line when a certain event occurs. Control is transferred as if a GOSUB statement had been made to the trap routine starting at the specified line number.

After a trap routine has serviced the event, the program resumes at the point of the event trap.

Events which can be trapped are :

- \* Detection of certain keystrokes (ON KEY)
- \* Time passage (ON TIMER)
- \* Emptying of the background music queue (ON PLAY)
- \* Joystick trigger activation (ON STRIG)
- \* Receipt of characters from a communications port COM.

Event trapping is controlled by the statements:

<event specifier> ON to turn on trapping

<event specifier> OFF to turn off trapping

<event specifier> STOP to temporarily turn off trapping

Event specifier is one of the following:

KEY (n)

KEY (n) ON is not the same statement as KEY ON.

KEY(n) ON sets an event trap for the specified key;

KEY ON displays the values of all the function keys on the twenty-fifth line of the screen .

When the GW-BASIC Interpreter is in direct mode, function keys maintain their standard meanings.

When a key is trapped, it is not remembered, so you cannot afterwards use INPUT or INKEY\$ to find which key caused the trap. To assign different functions to particular keys, set up a different subroutine for each key, rather than assign the various functions within a single subroutine.

## TIMER

ON TIMER(n) (n) is a numeric expression representing a number of seconds since the previous midnight. ON TIMER statements perform background tasks at defined intervals.

## PLAY

ON PLAY(n) (n) is a number of notes left in the music buffer. The statement retrieves more notes from the background music queue, to allow continuous background music during a program.

## STRIG (n)

n is the number of the joystick trigger. For most machines, n is zero to two.

For details of STRIG as a function, see Section 5.

## COM (n)

where n is the number of the communications channel. The n is the same device referred to in a COMn: statement. The COM channels are numbered 1 through n, where n is implementation dependent.

Typically, the COM trap routine will read an entire message from the COM port before returning. The COM trap is not recommended for single character messages because at high baud rates the overhead of trapping and reading for each character may allow the interrupt buffer for COM to overflow.

## ON GOSUB Statement

This sets up a line number for the specified event trap. The format is:

ON <event specifier> GOSUB <line number>

A line number of zero disables trapping for that event.

When an event is ON GW-BASIC checks whether the event has occurred each time it starts a new statement (except where the specified line number is zero).

When an event is OFF, no trapping takes place, and the event is not remembered even if it occurs.

When an event is stopped (<event specifier> STOP), no trapping takes place, but the event is remembered so an immediate trap takes place when an <event specifier> ON statement is executed.

A trap automatically causes a STOP on its event, so recursive traps never occur. A return from the trap routine automatically carries out an ON statement unless an explicit OFF was performed inside the trap routine.

All trapping is automatically disabled when an error trap occurs.

## RETURN Statement

When an event trap is in effect, a GOSUB statement is carried out as soon as the specified event occurs. For example, the statement

```
ON KEY GOSUB 1000
```

directs the program to line 1000 as soon as the key is used. If a simple RETURN statement is made at the end of this subroutine, program control returns to the statement after the one where the trap occurred. The corresponding GOSUB return address is cancelled.

GW-BASIC also includes RETURN <line number> which lets processing resume at a specified line. Use this option carefully. For example:

```
10 ON KEY GOSUB 1000
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE
200 REM PROGRAM RESUMES HERE
1000 'FIRST LINE OF SUBROUTINE
.
.
.
1050 RETURN 200
```

If the key is activated while the FOR/NEXT loop is running the subroutine is performed, but program control returns to line 200 instead of completing the FOR/NEXT loop. The original GOSUB entry is cancelled by the RETURN statement, and any other GOSUB, WHILE, or FOR (for example, an ON STRIG statement) active at the time of the trap remains active. The current FOR context also remains active, and a "FOR without NEXT" error may result.



# WRITING PROGRAMS USING THE GW-BASIC EDITOR



# Writing Programs Using the GW-BASIC Editor

3

GW-BASIC provides two ways to enter and edit text. Issuing an EDIT command (see Section 5) places you in edit mode; this is a useful way to alter individual lines of an existing program.

Alternatively, the full screen editor is available at any time between the Ok prompt and RUNning a program. This section shows how to use the full screen editor to write and edit programs.

## 3.1 Full Screen Editor

With the full screen editor the screen shows immediately what you have entered. Special keys for cursor movement, character insertion and deletion, and line or screen erasure allow speedy entering and correction of program text. With a program listing on screen, you can easily move the cursor to the required program lines, edit them, and make the changes permanent by pressing **RETURN**.

When input processes are directed from the screen, the full-screen editor features can be used in response to INPUT and LINE INPUT statements.

### Writing Programs

The full screen editor is available at any time after the interpreter's Ok prompt until you issue a RUN command. Any line of text you enter is processed by the editor. Those beginning with a number are considered as program statements.

Logical lines can be extended over more than one physical line by typing beyond the last column of the screen. The editor "wraps" the logical line on to the next physical line. Pressing **RETURN** signals the end of the logical line, and passes the entire logical line to GW-BASIC.

The editor deals with program statements by:

- \* Adding a new line to the program. This happens if the line number is valid (zero through 65529) and at least one non-blank character follows the line number.
- \* Modifying an existing line. If the line number matches that of an existing line in the program, the existing line is replaced with the text of the new line.
- \* Deleting an existing line. This occurs if the line contains only the line number, and the number matches that of an existing line.
- \* Passing the statements to the command scanner for interpretation and execution.

- \* Signalling an error. "Undefined line" means that an attempt has been made to delete a non-existent line.
- \* "Out of memory". The program memory is exhausted, and an unsuccessful attempt has been made to add a line to the program.

More than one statement can be on a line, but they must be separated by a colon (:). The colon need not be surrounded by spaces.

## Editing Programs

Use the LIST command to display a program or range of lines on screen. Then modify the text by moving the cursor where needed to:

- \* Type over existing characters
- \* Delete characters to the right of the cursor
- \* Delete characters to the left of the cursor
- \* Insert characters
- \* Add characters to the end of the logical line

To make text changes permanent, put the cursor anywhere on the relevant logical line and press `RETURN` .

The following table lists the available control keys and their effects.

Hex. Code	Control Key	Function
01	Ctrl-A	Enter edit mode
02	Ctrl-B	Move cursor to start of previous word
03	Ctrl-C	Break
04	Ctrl-D	No effect
05	Ctrl-E	Truncate line (clear text to end of logical line)
06	Ctrl-F	Move cursor to start of next word
07	Ctrl-G	Beep
08	Ctrl-H	Backspace, deleting characters passed over
09	Ctrl-I	Tab (8 spaces)
0A	Ctrl-J	Linefeed
0B	Ctrl-K	Move cursor to home position
0C	Ctrl-L	Clear window
0D	Ctrl-M	Carriage return (enter current logical line)
0E	Ctrl-N	Append to end of line
0F	Ctrl-O	Suspend or restart program output
10	Ctrl-P	No effect
11	Ctrl-Q	Restart suspended program
12	Ctrl-R	Toggle insert/typeover mode
13	Ctrl-S	Suspend program
14	Ctrl-T	Toggle function label key display
15	Ctrl-U	Clear logical line
16	Ctrl-V	No effect
17	Ctrl-W	Delete word
18	Ctrl-X	Display previous program line.
19	Ctrl-Y	Display following program line.
1A	Ctrl-Z	Clear to end of window
1C	→	Cursor right
1D	←	Cursor left
1E	↑	Cursor up
1F	↓	Cursor down (underscore)
7F	Ctrl-DEL	Delete character at cursor

### ***Commonly Used Commands***

GW-BASIC provides you with the facility to action some of its more commonly used commands by holding down the ALT key and pressing another key. The list below shows the commands which can be actioned by this method.

<b>Keystroke</b>	<b>Command</b>
ALT-A	AUTO
ALT-B	BSAVE
ALT-C	COLOR
ALT-D	DELETE
ALT-E	EDIT
ALT-F	FOR
ALT-G	GOTO
ALT-H	HEX\$
ALT-I	INPUT
ALT-J	J
ALT-K	KEY
ALT-L	LOCATE
ALT-M	MID\$
ALT-N	NEXT
ALT-O	OPEN
ALT-P	PRINT
ALT-Q	Q
ALT-R	RUN
ALT-S	SCREEN
ALT-T	THEN
ALT-U	USING
ALT-V	VAL
ALT-W	WIDTH
ALT-X	XOR
ALT-Y	Y
ALT-Z	Z

## Logical line Definition with INPUT

A logical line normally consists of all the characters on each of the component physical lines. To allow for forms input, INPUT and LINE INPUT statements act differently; the logical line is restricted to characters actually typed or passed over by the cursor. Insert and delete modes move only characters within the logical line, and delete mode reduces the size of the line.

Insert mode increases the logical line. If the characters moved write over non-blank characters on the same physical line - but not part of the logical line - the non-blank characters not part of the logical line are preserved, and the characters at the end of the logical line are thrown out. This preserves labels which existed before the INPUT statement.

## Editing Lines with Syntax Errors

When a syntax error occurs during a program run, GW-BASIC prints the line containing the error and enters direct mode. Correct the error, enter the change, and rerun the program.

When a line is modified, all files are closed, and all variables are lost. To examine the contents of variables just before the syntax error occurred, print the values before modifying the program line. The STOP and END commands will also enter direct mode without erasing variable values or closing files.

# WORKING WITH FILES AND DEVICES



# Working with files and devices

## 4

This section shows how files and devices are used and addressed in GW-BASIC, and the way information is input and output through the system.

## 4.1 Default device

When a file specification is given in commands or statements such as FILES, OPEN and KILL, the default (current) disk drive is the same as the default in MS-DOS before GW-BASIC was invoked.

## Device-independent Input/Output

## 4.2

GW-BASIC statements, commands and functions (listed below) provide device-independent input/output, and allow a flexible approach to data processing, as the syntax for access is the same for any device.

BLOAD	LOF
BSAVE	MERGE
CHAIN	OPEN
CLOSE	POS
EOF	PRINT
GET	PRINT USING
INPUT	PUT
INPUT\$	RUN
LINE INPUT	SAVE
LIST	WIDTH
LOAD	WRITE
LOC	

Individual descriptions are in Section 5.

## Working With Pathnames in BASIC

BASIC can create, change, and remove paths, using the commands MKDIR, CHDIR, and RMDIR. For example

```
MKDIR "ACCOUNTS"
```

creates a new directory ACCOUNTS in the working directory of the current drive

```
CHDIR "B:EXPENSES"
```

changes the current directory on B: to EXPENSES

```
RMDIR "CLIENTS"
```

deletes an existing directory CLIENTS as long as it was empty of all files with the exception of "." and "..".

For further information on handling paths in BASIC, see the CHDIR, MKDIR, and RMDIR Statements in Section 5.

## Re-direction of standard input and standard output

## 4.4

BASIC can be re-directed to read from standard input and write to standard output by providing the input and output filenames on the command line as follows:

GW-BASIC [program name] [<input file] [>output file]

You must enter the characters < and > here; they are not part of the syntax conventions used in this manual. If you put >> before the output file name, the output is added to that file.

- \* When re-directed, all INPUT, LINE INPUT, INPUT\$, and INKEY\$ statements read from the input file.
- \* If the program does not specify a file number in a PRINT statement, output is redirected to the declared output file instead of the standard output device, the screen.
- \* Error messages go to standard output.
- \* File input from "KYBD:" still reads from the keyboard.
- \* File output to "SCRN:" still appears on the screen.
- \* BASIC continues to trap keys from the keyboard when the ON KEY(n) statement is used.
- \* The printer echo key does not cause LST: echoing if standard output has been re-directed.
- \* Using CTRL-C causes BASIC to close any open files, issue the message "Break in line <line-number>" to standard output, and exit BASIC.
- \* When input is redirected, BASIC reads from this source until an end-of-file character is detected.

You can test this with the EOF function. If the file is not ended by a CTRL-Z, or a BASIC input statement tries to read past end-of-file, any open files are closed, the message "Read past end" is written to standard output, and BASIC terminates.

## Examples

```
GW-BASIC MYPROG > DATA.OUT
```

Data read by INPUT and LINE INPUT continues to come from the keyboard. Data output by PRINT goes into the file DATA.OUT.

```
GW-BASIC MYPROG < DATA.IN
```

Data read by INPUT and LINE INPUT comes from DATA.IN. Data output by PRINT continues to go to the screen.

```
GW-BASIC MYPROG < MYINPUT.DAT > MYOUTPUT.DAT
```

Data read by INPUT and LINE INPUT now comes from the file MYINPUT.DAT and data output by PRINT goes into MYOUTPUT.DAT.

```
GW-BASIC MYPROG < \SALES\JOHN\TRANS.  
\SALES\SALES.DAT
```

Data read by INPUT and LINE INPUT now comes from the file SALES JOHN TRANS. Data output by PRINT is appended to the file SALES SALES.DAT.

This section explains file I/O (input and output) procedures for those new to BASIC, and forms a useful checklist in case of file-related errors in programs.

## Program file commands

The following commands and statements are used in program file manipulation. File specifications can include a device and a pathname.

### **SAVE <filespec> {[.A]P}**

Writes the program currently in memory to the specified file. Option A writes the program as a series of ASCII characters. With option P, BASIC encodes the file in a read-protected format.

### **LOAD <filespec> [R]**

Loads the program from file into memory. The optional R runs the program immediately. LOAD deletes the current contents of memory and closes all files before loading. Option R keeps open data files open, so that programs can be chained or loaded in sections with access to the same data files. (LOAD FILESPEC>,R and RUN FILESPEC),R are equivalent.)

### **RUN <filespec> [,R]**

Loads the program from file into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. Option R keeps all open data files open. (RUN <filespec>,R and LOAD <filespec>,R are equivalent.)

## **MERGE <filespec>**

Loads the program from file into memory but does not delete current memory contents.

The file specification must have been saved in ASCII format. Program line numbers in the file are merged with line numbers in memory.

Where two lines have the same number, only the line from the file program is saved. After a MERGE command the *merged* program resides in memory, and BASIC returns to command level.

## **CHAIN [MERGE] <filespec> [, [<line number exp>] [,ALL] [,DELETE <range> ]]**

<line number expression> is the line number giving the starting point for the called program. The command passes control to the named program, together with the use of the variables and their current values. Options are to start the new program on a specified line, delete some lines, or transfer the values of only some of the variables.

## **KILL <filespec>**

Deletes the file from the disk. <filespec> can be a program file or a sequential or random access data file.

## **NAME <old filespec> AS <new filespec>**

Changes the name of a file. Use NAME AS <filespec> with program files, random access files, or sequential files. Pathnames are not allowed.

## ***Protecting Program Files***

To save a program in an encoded binary format, use the SAVE command with the "Protect" option: SAVE "MYPROG",P

The program cannot then be listed or edited, so it is useful to save an unprotected copy of the program for these purposes.

A BASIC program can create and use two kinds of disk data files; sequential files and random access files.

### Sequential files

Sequential files are easier to create than random access but are limited in flexibility and speed when locating data. The data written to a sequential file is a series of ASCII characters stored sequentially in the order sent.

The data is read back sequentially.

Statements and functions used with sequential data files are:

OPEN

WIDTH

PRINT#

PRINT USING#

WRITE#

INPUT#

INPUT\$

LINE INPUT#

EOF

LOC

LOF

CLOSE

## Creating a sequential file

Program 1 creates a sequential DATA file from information input at the keyboard.

Program 1

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$ = "DONE" THEN END
30 INPUT "DEPARTMENT";DEPT$
40 INPUT "DATE HIRED";HIREDATES
50 PRINT#1,N$,"";DEPT$,"";HIREDATES
60 PRINT
70 GOTO 20
RUN

NAME? SAMUEL GOLDWYN
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? MARVIN HARRIS
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? DEXTER HORTON
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/81

NAME? STEVEN SISYPHUS
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/81

NAME? etc.
```

Program steps to create a sequential file and access data in it:

- \* OPEN the file in "O" mode.
- \* Write data to the file using the PRINT# statement.  
(Write # can be used instead.)
- \* To access the data in the file, CLOSE the file and reopen it in "I" mode.
- \* Use the INPUT# statement to read data from the sequential file into the program.

## Reading data from a sequential file

Program 2 accesses the file DATA created in Program 1 and displays the name of everyone hired in 1981.

Program 2

```
10 OPEN "I", #1, "DATA"  
20 INPUT #1, N$, DEPT$, HIREDATE$  
30 IF RIGHT$(HIREDATE$, 2) = "81" THEN PRINT N$  
40 GOTO 20  
RUN  
  
DEXTER HORTON  
STEVEN SISYPHUS  
Input past end in 20
```

Program 2 reads, sequentially, every item in the file, and prints the names of employees hired in 1981. When all the data has been read, line 20 causes an INPUT PAST END error. To avoid this, use the WHILE...WEND construct. This tests for end of file with the EOF function.

Revised Program 2

```
10 OPEN "I", #1, "DATA"  
15 WHILE NOT EOF(1)  
20 INPUT #1, N$, DEPT$, HIREDATE$  
30 IF RIGHT$(HIREDATE$, 2) = "81" THEN PRINT N$  
40 WEND
```

A program which creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement below would write numeric data to the file without explicit delimiters.

```
PRINT#1, USING "#####.##,"; A, B, C, D
```

The commas at the end of the format string separate the items in the disk file. To make commas appear in the field as delimiters between variables, use the WRITE statement. For example, the statement

```
WRITE 1, A, B$
```

would write these two variables to the file with commas delimiting them.

When used with a sequential file, the LOC function returns the number of sectors (128-byte blocks) written to or read from the file since it was opened.

## Adding data to a sequential file

Use the append mode "A". If the file doesn't already exist, the open statement works exactly as if output ("O") mode had been specified. Opening a sequential file in O mode destroys its current contents. Append mode does not destroy existing data.

### Program 3

```
110 OPEN "A",#1,"FOLKS"  
120 REM ADD NEW ENTRIES TO FILE  
130 INPUT "NAME";N$  
140 IF N$="" THEN 200 'CARRIAGE RETURN  
EXITS INPUT LOOP  
150 LINE INPUT "ADDRESS?";ADDR$  
160 LINE INPUT "BIRTHDAY?";BIRTHDATE$  
170 PRINT#1,N$  
180 PRINT#1,ADDR$  
190 PRINT#1,BIRTHDATE$  
200 GOTO 120  
210 CLOSE 1
```

## Random access files

Random access files need more program steps than sequential files, but have advantages. They need less room on disk because BASIC stores them in a packed binary format. A sequential file is stored as a series of ASCII characters.

More importantly, they store data in distinct numbered records. This allows them to access data randomly - they can retrieve data from anywhere on the disk without reading right from the beginning of the file.

Statements and functions used with random access files are:

OPEN	CVD
FIELD	CVI
GET	CVS
LOC	MKS\$
LOF	MKD\$
LSET	MKI\$
RSET	
PUT	
CLOSE	

## Creating a random access file

Program 4 takes information input at the terminal and writes it to a random access file. The PUT statement writes each record to the file. The two-digit code input in line 30 becomes the record number.

Program 4

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";PERSON$
50 INPUT "AMOUNT";AMOUNT
60 INPUT "PHONE";TELELPHONES$
65 PRINT
70 LSET N$ = PERSON$
80 LSET A$ = MKS$(AMOUNT)
90 LSET P$ = TELELPHONES$
100 PUT #1,CODE%
110 GOTO 30
```

Program steps to create a random access file:

- \* **OPEN the file for random access ("R" mode).** Default record length is 128 bytes, unless set to another value with the /I/S: switches when starting BASIC. This example specifies a record length of 32 bytes: value with the /I/S: switches when invoking BASIC.

```
OPEN "R", 1, "FILE", 32
```

- \* **Use the FIELD statement to allocate space in the random buffer for the variables to be written to the random file.**

```
FIELD #1, 20 AS N$, 4 AS ADDR$, 8 AS P$
```

- \* **Use LSET to move the data into the random access buffer.** Make numeric values into strings - use the make functions; MKI\$ to make an integer value into a string, MKS\$ to make a single precision value into a string, and MKD\$ to make a double precision value into a string. For example

```
LSET N$ = X$  
LSET ADDR$ = MKS$(AMT)  
LSET P$ = TEL$
```

- \* **Write the data from the buffer to the disk using the PUT statement.** For example

```
PUT #1, CODE%
```

### Note

Do not use a fielded string variable in an INPUT or LET statement - this causes the variable to be redeclared. BASIC will no longer associate the variable with the file buffer, but with the new program variable.

## Accessing a random access file

Program 5 accesses the random access file "FILE" created in Program 4. To retrieve information, enter its three-digit code at the terminal.

Program 5

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Program steps to access a random access file:

- \* OPEN the file in R mode. For example

```
OPEN "R", 1,"FILE",32
```

- \* Use the FIELD statement to allocate space in the random access buffer for the variables to be read from the file. For example

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```

In programs carrying out input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

- \* Use the GET statement to move the desired record into the random access buffer. For example

```
GET #1,CODE%
```

The program can now access data in the buffer.

Numeric values converted to strings by MKS\$, MKD\$ or MKI\$ statements must be converted back to numbers with the "convert" functions; CVI for integers, CVS for single precision values, and CVD for double precision values. MKI\$ and CVI mirror each other; MKI\$ converts a number into a format for storage in random files, CVI converts the random file storage into a form suitable for the program. For example:

```
RINT N$  
PRINT CVS(A$)
```

When used with random access files, the LOC function returns the "current record number"; that is, the last record number used in a GET or PUT statement.

### Example

```
IF LOC(1) > 50 THEN END
```

This stops a program if the current record number of file#1 is greater than 50.

## Random file operations

Program 6 is an inventory program which illustrates random file access.

Program 6

```
120 OPEN "R", #1, "INVEN.DAT", 39
125 FIELD #1, 1 AS F$, 30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT "1, INITIALISE FILE"
140 PRINT "2, CREATE A NEW ENTRY"
150 PRINT "3, DISPLAY INVENTORY FOR ONE PART"
160 PRINT "4, ADD TO STOCK"
170 PRINT "5, SUBTRACT FROM STOCK"
180 PRINT "6, DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT "FUNCTION":FUNCTION
225 IF (FUNCTION < 1) OR (FUNCTION > 6) THEN
PRINT "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM ** BUILD NEW ENTRY **
260 GOSUB 840
270 IF ASC(F$) < > 255 THEN INPUT "OVERWRITE":
ADDR$:
IF ADDR$ < > "Y" THEN RETURN
280 LSET F$ = CHR$(0)
290 INPUT "DESCRIPTION":DESCRIPTION$
300 LSET D$ = DESCRIPTION$
310 INPUT "QUANTITY IN STOCK":QUANTITY%
320 LSET Q$ = MKI$(QUANTITY%)
330 INPUT "REORDER LEVEL":REORDER%
340 LSET R$ = MKI$(REORDER%)
350 INPUT "UNIT PRICE":PRICE
360 LSET P$ = MKS$(PRICE)
370 PUT #1, PART%
380 RETURN
390 REM ** DISPLAY ENTRY **
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
```

```

450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":
RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD";
ADDITIONAL%
520 Q% = CVI(Q$) + ADDITIONAL%
530 LSET Q$ = MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":
RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";LESS%
610 Q% = CVI(Q$)
620 IF (Q%-LESS%) < 0 THEN PRINT "ONLY";Q%;
"IN STOCK":GOTO 600
630 Q% = Q% - LESS%
640 IF Q% = < CVI(R$) THEN PRINT
"QUANTITY NOW";Q%;"REORDER LEVEL";CVI(R$)
650 LSET Q$ = MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
710 GET#1,I
720 IF CVI(Q$) < CVI(R$) THEN PRINT D$;
"QUANTITY";CVI(Q$) TAB(50) "REORDER
LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART% < 1) OR (PART% > 100) THEN PRINT
"BAD PART NUMBER":GOTO 840 ELSE GET#
1,PART%:RETURN
890 END
900 REM INITIALISE FILE
910 INPUT "ARE YOU SURE";CONFIRM$:IF
CONFIRM$ <> "Y" THEN RETURN

```

```
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

The record number acts as the part number. It is assumed the inventory contains no more than 100 different part numbers. Lines 900-960 initialise the data file by writing CHR\$(255) as the first character of each record. In lines 270 and 500 this determines whether an entry already exists for that part number.

Lines 130-220 display the various inventory functions provided by the program. When you enter the required function number, line 230 branches to the appropriate subroutine.



# COMMANDS, FUNCTIONS AND STATEMENT SPECIFICATIONS



# Dictionary of commands, functions and statements

5

This section shows in more detail the commands, functions and statements used in GW-BASIC.

## ABS Function

Returns the absolute value of the expression X.

### Syntax

ABS(X)

### Example

```
PRINT ABS(7*(-5))
```

returns

```
35
```

## ASC Function

Returns a numerical value representing the ASCII code for the first character of the string X\$. (See Appendix for ASCII codes.)

### Syntax

ASC(X\$)

If X\$ is null, an “Illegal function call” error is returned.

### Example

```
10 X$="TEST"  
20 PRINT ASC(X$)
```

returns

```
84
```

See the CHR\$ function for details on ASCII-to-string conversion.

## ATN Function

Returns the arctangent of X, where X is in radians. Result is in the range  $-\pi/2$  to  $\pi/2$  radians.

### Syntax

ATN(X)

The expression X can be any numeric type, but the default evaluation of ATN is performed in single precision. This may be overridden if the /D switch is used when invoking GW-BASIC 2.0.

### Example

```
10 LET X = 3
20 PRINT ATN(X)
```

returns

```
1.249046
```

## AUTO Command

Automatically generates line numbers during program entry.

### Syntax

AUTO [<line number> [, <increment> ]]

AUTO begins numbering at <line number> and increments each subsequent line number by the specified amount. The default for both values is 10. If line number is followed by a comma but the increment is not specified, the last one specified in an AUTO command is assumed.

If a line number is already being used an asterisk is printed after the number to warn that any input will replace the existing line. Typing a carriage return immediately after the asterisk saves the existing line and generates the next line number.

If the cursor is moved to another line on the screen, numbering resumes there.

AUTO is terminated by typing a CTRL-C. The line where the CTRL-C is typed is not saved. Microsoft GW-BASIC returns to command level after the CTRL-C is used.

### Example

```
AUTO 100,50
```

Generates line numbers 100,150, 200 and so on.

```
AUTO
```

Generates line numbers 10, 20, 30, 40 and so on.

## BEEP Statement

Sounds the speaker (via the ASCII bell character)

### Syntax

BEEP

This has the same effect as PRINT CHR\$(7) in nongraphics versions of MS-BASIC.

### Example

```
20 IF X < 20 THEN BEEP
```

Beep sounds out when X is less than 20

## BLOAD Command

Loads a specified memory image file into memory from disk.

### Syntax

BLOAD <filespec> [, <offset>]

The filename can be one to eight characters long.

<offset> is a numeric expression returning an unsigned integer in the range zero to 65535. This is the loading offset address in the segment declared by the last DEF SEG statement. If no DEF SEG statement is given the GW-BASIC data segment is used as the default. If the offset is omitted, the segment address and offset contained in the file are used.

## Warning

As BLOAD does not perform an address range check it is possible to load a file anywhere in memory. Do not load over the operating system.

## Example

```
20 DEF SEG = &H6000 'Set segment to 6000 Hex
30 BLOAD"PROG1",&HF000 'Load PROG1
```

The segment address is set at 6000 Hex and loads PROG1 at F000.

## BSAVE Command

Transfers the contents of the specified area of memory to disk.

### Syntax

BSAVE <filespec> , <offset> , <length>

The filename must be one to eight characters long.

<offset> is a numeric expression returning an unsigned integer in the range zero to 65535. It is the saving offset address in the segment declared by the last DEF SEG statement.

<length> is a numeric expression returning an unsigned integer in the range one to 65535, and is the length in bytes of the memory image file being saved.

BSAVE can save data or programs can be saved as memory image files on disk.

If the offset, or the length are omitted, there is a "Bad file name" message and the save is terminated. A DEF SEG statement must be made before the BSAVE. The last known DEF SEG address is used for the save.

## Example

```
10 'Save PROG1
20 DEF SEG = &H6000
30 BSAVE"PROG1",&HF000,256
```

This example saves 256 bytes starting at 6000:F000 in the file PROG1.

## CALL Statement

Calls an assembly language subroutine or a compiled routine written in another high level language.

### Syntax

CALL <variable name> [( <argument list> )]

<variable name> contains a starting address in memory of the subroutine. It must not be an array variable name.

The argument list, of variables only, contains the arguments passed to the external subroutine.

The CALL statement is one way to transfer program flow to the external subroutine. The USR function can also be used. See section 16.

### Example

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
```

## CALLS Statement

This is similar to CALL, except that the segmented addresses of all arguments are passed. Use CALLS when accessing routines written with the FORTRAN calling convention. All FORTRAN parameters are call-by-reference segmented addresses.

CALLS uses the segment address defined by the most recent DEF SEG statement to locate the routine being called.

## CDBL Function

Returns X as a double precision number.

### Syntax

CDBL(X)

### Example

```
10 LET PI = 22/7
20 PRINT PI,CDBL(PI)
```

returns

3.142857

3.142857074737549

## CHAIN Statement

Calls a program and may pass variables to it from the current program.

### Syntax

CHAIN [MERGE] <filespec> [, [ <line number exp> ]  
[,ALL][,DELETE <range> ]]

MERGE can bring a subroutine into the GW-BASIC program as an overlay. (See DELETE below). The current program and the called program are merged (see MERGE Command). The called program must be an ASCII file.

<filespec> is a conforming string expression for disk filenames and device specifications.

<line number exp> is a line number (or an expression evaluating to a line number) giving the starting point for the called program. It is not affected by a RENUM command. To omit it, put a comma instead: execution begins at the first line.

ALL passes every variable in the current program to the called program. If omitted, both programs must contain a COMMON statement listing the variables that are passed.

After DELETE a new overlay can be brought in.

RENUM affects the line numbers in the range.

## Example 1

Two string arrays are dimensioned, and declared as common variables. When Program 1 gets to line 90, it chains to Program 2 which loads the B\$ array. At line 90 of PROG2, control chains back to the first program at line 100. This process can be seen through the descriptive text that prints as the programs run.

### PROGRAM 1

```
10 REM THIS PROGRAM DEMONSTRATES
   CHAINING USING COMMON TO PASS VARIABLES.
20 REM SAVE THIS MODULE ON DISK AS "PROG1"
   USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2)="VALUES BEFORE CHAINING."
70 B$(1)="  "
80 B$(2)="  "
90 CHAIN "PROG2"
100 PRINT
110 PRINT B$(1)
120 PRINT
130 PRINT B$(2)
140 PRINT
150 END
```

## PROGRAM 2

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY  
ONLY BE EXECUTED ONCE.  
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.  
30 REM SAVE THIS MODULE ON THE DISK  
AS "PROG2" USING THE A OPTION.  
40 COMMON A$(),B$()  
50 PRINT  
60 PRINT A$(1);A$(2)  
70 B$(1) = "NOTE HOW THE OPTION OF SPECIFYING A  
STARTING LINE NUMBER"  
80 B$(2) = "WHEN CHAINING AVOIDS THE DIMENSION  
STATEMENT IN 'PROG1'."  
90 CHAIN "PROG1",100  
100 END
```

### Example 2

This shows the MERGE, ALL, and DELETE options. After A\$ is loaded in Program A control chains to line 1010 of Program B. At Program B's line 1040 it chains to line 1010 of the Program C keeping all variables and deleting all previous program's lines. Control passes to Program C.

```

10 REM THIS PROGRAM DEMONSTRATES
CHAINING USING THE MERGE, ALL, AND DELETE
OPTIONS.
20 REM SAVE THIS MODULE ON THE DISK AS
"MAINPRG".
30 A$="MAINPRG"
40 CHAIN MERGE "OVRLAY1",1010,ALL
50 END

1000 REM SAVE THIS MODULE ON THE DISK AS
"OVRLAY1" USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO OVRLAY1."
1020 A$="OVRLAY1"
1030 B$="OVRLAY2"
1040 CHAIN MERGE "OVRLAY2",1010,ALL,
DELETE 1000-
1050
1050 END

1000 REM SAVE THIS MODULE ON THE DISK AS
"OVRLAY2" USING THE A OPTION.
1010 PRINT A$; "HAS CHAINED TO"; B$;"."
1020 END

```

### Note

The CHAIN statement with MERGE leaves the files open and keeps the current OPTION BASE setting.

If MERGE is omitted, CHAIN does not keep variable types or user-defined functions for use by the chained program. Any DEF SNG/DBL/STR or DEF FN statements containing shared variables must be restated in the chained program.

When using the MERGE option, user-defined functions are placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions are undefined after the merge is complete.

## CHDIR Statement

Changes the current operating directory.

### Syntax

CHDIR Pathname

Pathname is a string of less than 128 characters specifying the name of the next directory. It works like the MS-DOS command CHDIR.

### Example

```
CHDIR "SALES"
```

This makes SALES the current directory.

```
CHDIR "B:USERS"
```

This changes the current directory to USERS on drive B. It does not change the default drive to B.

See also the MKDIR and RMDIR statements.

## CHR\$ Function

Returns a string whose one character has the ASCII value of I.

### Syntax

CHR\$(I)

CHR\$ may be used to send a special character to the screen or printer. For instance, the BELL character (CHR\$(7)) could be sent as a preface to an error message, or a form feed (CHR\$(12)) sent to clear a terminal screen and return the cursor to the home position.

### Example

```
PRINT CHR$(66)
```

returns

```
B
```

See the ASC function for details on ASCII-to-numeric conversion.

## CINT Function

Returns X as an integer by rounding the fractional portion.

### Syntax

CINT(X)

If X is not in the range - 32768 to 32767, an "Overflow" error occurs.

### Example

```
PRINT CINT(45.67)
```

returns

```
46
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. The FIX and INT functions also return integers.

## CIRCLE Statement

Draws an ellipse or circle with the specified centre and radius.

### Syntax

**CIRCLE** [**STEP**] ( <xcentre> , <ycentre> ) , <radius>  
[ , <colour> [ , <start> , <end> [ , <aspect> ] ] ]

[**STEP**] ( <xoffset> , <yoffset> ) makes the specified centre and ycentre coordinates relative to the "most recent point", instead of absolute, mapped coordinates.

<xcentre> is the x coordinate for the centre of the circle;  
<ycentre> is the y coordinate for the centre of the circle.  
<radius> is the radius of the circle in the current logical coordinate system. <colour> is the numeric symbol for the colour desired (see **COLOR** Statement). The default colour is the foreground colour.

<start> and <end> are the angles in radians of the start and end of the ellipse. The range is  $-2\pi$  through  $2\pi$ . If they are negative, the ellipse is connected to the centre point with a line, and the angles treated as if they were positive. Note that this is different from adding  $2\pi$ . The start angle may be less than the end angle.

<aspect> is the ratio of the x radius to the y radius. Default ratios depend on the machine being used. When they are specified for the corresponding screen mode, a round circle is drawn.

If the aspect ratio is less than one, the radius given is the x radius. If it is greater than one, the y radius is given.

The centre of the circle is the last point referenced.

Coordinates outside the screen or viewport can be supplied.

## Example

If the last point referenced was (10,10), STEP references a point offset 10 from the current x coordinate and offset 5 from the current y coordinate, that is, the point (20, 15). If the last point plotted was 100,50 then,

```
CIRCLE (100,100) ,50
```

and

```
CIRCLE STEP (0,100) ,50
```

both draw a circle at 100,100 with radius 50. The first example uses absolute notation and the second relative.

## CLEAR Statement

Sets all numeric variables to zero, all string variables to null, closes all open files; also sets the end of memory and the amount of stack space.

### Syntax

```
CLEAR [, [ <expression1 > ] [, <expression2 > ]]
```

<expression1 > is the highest memory location available.

<expression2 > sets aside stack space for GW-BASIC.

The default is 768 bytes or one-eighth of the available memory, whichever is smaller.

CLEAR also clears all COMMON variables, releases all disk buffers and resets all DEF FN and DEF SNG/DBL/STR statements.

### Examples

```
CLEAR  
CLEAR , 32768  
CLEAR ,,2000  
CLEAR ,32768,2000
```

## CLOSE Statement

Concludes I/O to a file. CLOSE complements the OPEN statement.

### Syntax

`CLOSE [[#] <file number> [, [#] <file number...> ]]`

<file number> is the opening number. CLOSE with no arguments closes all open files.

After CLOSE the file number does not belong to any file and can be used for any unopened file. You can reopen a file using the same or a different file number.

A CLOSE for a sequential output file writes the final buffer of output.

The SYSTEM, CLEAR, and END statements and the NEW and RESET commands close all files automatically.

### Example

```
CLOSE #1,#2
```

## CLS Statement

Erases contents of entire current screen.

### Syntax

`CLS`

The Clear Window key also clears the screen.

### Example

```
10 CLS' Clears the screen
```

## COLOR Statement

Selects the foreground, background, and border colours for the graphics display, and affects the colour of text. These colours are used by the PSET, PRESET, and LINE statements.

### Syntax

The syntax and purpose of the COLOR statement is machine dependent. Usually the form is : COLOR [`<parameter1>`] [`<parameter2>`] ...]

PARAMETERS are integers and the statement modifies the current default text, foreground or background colour. They can be omitted and the previous value is then kept.

Any parameters outside the numeric ranges specified for the machine lead to a "Illegal function call". Previous values are then retained.

Foreground and background colour may be the same so making displayed characters invisible.

## COM Statement

Enables or disables event trapping of communications activity on the specified port.

### Syntax

COM(`n`) ON  
COM(`n`) OFF  
COM(`n`) STOP

The parameter `n` is the number of the communications port. The range for `n` is specified by the implementor.

The COM ON statement enables communications event trapping by an ON COM statement. If you specify a non-zero line number in the ON COM statement while trapping is enabled, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, it executes the ON COM statement.

COM OFF disables communications event trapping. If an event takes place, it is not remembered.

COM STOP disables communications event trapping, but if an event occurs, it is remembered. If there is a subsequent COM ON statement, the remembered event will be successfully trapped.

### Example

This enables error trapping of communications activity on channel 1:

```
10 COM(1) ON
```

### Compiler/Interpreter Differences

See "ON COM Statement".

## COMMON Statement

May be used to pass variables to a chained program.

### Syntax

```
COMMON <list of variables>
```

COMMON is used with CHAIN. COMMON statements may appear anywhere in a program, though we recommend they appear at the start. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

With some BASIC languages the number of dimensions in an array can be included in the COMMON statement. GW-BASIC accepts this syntax but ignores the numeric expression. For instance, the following statements are both valid and are considered equivalent:

```
COMMON A()  
COMMON A(3)
```

The number in parentheses is the number of dimensions, not the dimensions themselves. The variable A(3) in this example might correspond to a DIM statement of DIM A(5,8,4).

## Example

```
100 COMMON A,B,C,D(),G$  
110 CHAIN "PROG3",10
```

## CONT Command

Continues the program after a Break has been typed or a STOP statement been made.

### Syntax

CONT

If the break came after a prompt from an INPUT statement, the program continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used with STOP for debugging. When stopped, intermediate values can be examined and changed using direct mode statements. The program may be resumed with CONT or a direct mode GOTO, which resumes at a specified line number. You may use CONT after an error but it is invalid if the program has been edited during the break.

## COS Function

Returns the cosine of X, where X is in radians.

### Syntax

COS(X)

The calculation of COS(X) is performed in single precision. This is unless the I/D switch is specified when BASIC is invoked and the argument that receives the value of the cosine is a double precision variable or (X) is specified a double precision number with the # sign.

### Example

```
10 X=2*COS(.4)
20 PRINT X
```

returns

```
1.842122
```

## CSNG Function

Returns X as a single precision number.

### Syntax

CSNG(X)

### Example

```
10 A# = 975.3421115#
20 PRINT A#, CSNG(A#)
```

returns

```
975.3421115      975.3421
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

## CSRLIN Function

Obtains the current line position of the cursor in a numeric variable.

### Syntax

CSRLIN

Use the POS function for returning the current column position.

### Example

```
10 y = CSRLIN 'Record current line.  
20 x = POS(0) 'Record current column.  
30 LOCATE 24,1  
40 PRINT "HELLO"  
50 LOCATE x,y 'Restore position to old line and column
```

## CVI, CVS, CVD Functions

Converts string values to numeric values.

### Syntax

CVI( <2-byte string> )

CVS( <4-byte string> )

CVD( <8-byte string> )

Numeric values read in from a random disk file must be converted from strings back into numbers.

CVI converts a 2-byte string to an integer.

CVS converts a 4-byte string to a single precision number.

CVD converts an 8-byte string to a double precision number.

## Example

```
.  
. .  
. .  
70 FIELD #1,4 AS N$, 12 AS B$, ...  
80 GET #1  
90 Y = CVS(N$)  
. .  
. .  
. .
```

See also MKI\$, MKS\$, MKD\$.

## DATA Statement

Stores the numeric and string constants accessed by the program's READ statement(s).

### Syntax

DATA <list of constants>

<list of constants> can contain numeric constants in any format; fixed-point, floating-point, or integer. Numeric expressions are not allowed. String constants must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces.

The variable type (numeric or string) in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements are not instructions and any number can be anywhere in a program. A DATA statement can contain as many constants as fit on a line. They must be separated by commas. READ statements access DATA statements in line order.

To reread a DATA statement from the start, use the RESTORE statement.

## DATE\$ Statement

Sets the current date. Complements the DATE\$ function, which retrieves the current date.

### Syntax

DATE\$ = <string expression>

The string expression must be in one of the following forms:

mm-dd-yy  
mm-dd-yyyy  
mm/dd/yy  
mm/dd/yyyy

### Example

```
10 DATE$="07-01-1983"
```

The current date is July 1, 1983.

## DATE\$ Function

Prints the date, calculated from the date set with the DATE\$ statement.

### Syntax

DATE\$

This function returns a ten-character string in the form mm-dd-yyyy. Mm is the month (01 through 12), dd is the day (01 through 31), and yyyy is the year (1980 through 2099).

### Example

```
10 PRINT DATE$
```

## DEF FN Statement

Defines and names a function you have written.

### Syntax

```
DEF FN <name> [( <parameter list> )] =  
<function definition>
```

Name must be a legal variable name. Preceded by FN, becomes the name of the function.

Parameter list includes the variable names in the function definition to be replaced when the function is called. The items in the list are separated by commas.

The Function definition describes the function. It is limited to one logical line. Variable names in this expression only define the function; they do not affect program variables of the same name. If a variable name used in a function definition appears in the parameter list, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values given in the function call.

Numeric or string expressions can be defined. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. The type specified in the function name must match the argument type. Otherwise there is a "Type Mismatch error".

Calling a function before a DEF FN statement leads to a "Undefined user function" error. DEF FN is illegal in the direct mode.

### Example

```
.  
100 DEF FNAB(X,Y) = X^3/Y^2  
.  
420 T = FNAB(I,J)  
.  
.
```

Line 100 defines the function FNAB. The function is called in line 420.

## DEFINT/SNG/DBL/STR Statements

Declares variable types as integer, single precision, double precision, or string.

### Syntax

DEF <type> <range(s) of letters>

where <type> is INT, SNG, DBL, or STR

Any variable names beginning with the letters in the range are considered the type of variable specified in the type portion of the statement. A type declaration character takes precedence over a DEFtype statement. See the section on "Variable Names and Declaration Characters,"

If no type declaration statements are encountered GW-BASIC assumes that all variables without declaration characters are single precision variables.

### Examples

```
10 DEFDBL L-P
```

All variables beginning with the letters L, M, N, O or P will be double precision variables.

```
10 DEFSTR A
```

All variables beginning with the letter A will be string variables.

```
10 DEFINT I-N,W-Z
```

All variables beginning with the letters I, J, K, L, M, N, W, X, Y or Z will be integer variables.

## DEF SEG Statement

Assigns the current segment address to be referenced by subsequent BLOAD, BSAVE, CALL, CALLS, or POKE statements or by the USR or PEEK functions.

### Syntax

DEF SEG [ = <address> ]

<address> is a numeric expression returning an unsigned integer in the range zero to 65535.

Entry of any value outside the address range zero through 65535 results in an "Illegal function call" and the previous value is retained. If the address is omitted, the GW-BASIC data segment is used. This is the initial default value.

DEF and SEG must be separated by a space. Otherwise the statement DEFSEG = 100 is taken to mean "assign the value 100 to the variable DEFSEG."

### Example

```
10 DEF SEG = &HB800 'Seg segment at B800 Hex
20 DEF SEG 'Restore segment to GW-BASIC data
   segment.
```

## DEF USR Statement

Specifies the starting address of an assembly language subroutine.

### Syntax

DEF USR[ <digit> ] = <integer expression>

<digit> is any from zero to nine and corresponds to the number of the USR routine whose address is being specified. If omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine.

Any number of DEF USR statements can appear in a program to redefine subroutine starting addresses.

### Example

```
200 DEF USR0=24000  
210 X=USR0(Y^2/2.89)
```

## DELETE Command

Deletes program lines.

### Syntax

```
DELETE {[ <line number> ] [- <line number> ]  
[ <line number> -]}
```

After DELETE GW-BASIC always returns to command level. If no line number is specified there is an "Illegal function call".

### Examples

```
DELETE 40
```

Deletes line 40.

```
DELETE 40-100
```

Deletes lines 40 through 100, inclusive.

```
DELETE -40
```

Deletes all lines up to and including line 40.

```
DELETE 40-
```

Deletes lines 40 through the end, inclusive.

## DIM Statement

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

### Syntax

DIM <list of subscripted variables>

If an array variable name is used without DIM the maximum value of the array's subscript is assumed to be 10. Using a subscript greater than the maximum specified causes a "Subscript out of range" error. The minimum value for a subscript is zero, unless otherwise specified with the OPTION BASE statement.

DIM sets all the elements of the specified numerical arrays to an initial value of zero and elements of string arrays to null strings. In theory, the maximum number of dimensions allowed in a DIM statement is 255. In reality, this number is impossible as the name and punctuation are also counted as spaces on the line, and the line itself has a limit of 255 characters.

If the default dimension (10) has already been established for an array variable in a DIM statement there will be an "Array already dimensioned" error. It is good practice to put the required DIM statements at the beginning of a program, outside of any processing loops.

### Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

```
.
```

```
.
```

```
.
```

## DRAW Statement

Draws an object defined by the subcommands described below.

**Note:** In order to use the DRAW statement you must select the correct screen. See SCREEN STATEMENT.

### Syntax

DRAW <string expression>

DRAW combines many graphics statements into the Graphics Macro Language. This defines a set of characteristics describing a particular image. These include motion, colour, angle, and scale factor.

Movement is from the current graphics position. This is usually the coordinate of the last graphics point plotted with another GML command. The current position is the centre of the screen when a program is run.

Prefix commands can precede movement commands;

B Move but don't plot any points

N Move and return to original position

The following commands specify cursor movement in units. Their size can be modified by the S command. The default unit size is one point. If no argument is supplied, the cursor is moved one unit.

U [<n>] Move up (scale factor \*n) points

D [<n>] Move down

L [<n>] Move left

R [<n>] Move right

E [<n>] Move diagonally up and right

H [<n>] Move diagonally up and left

G [<n>] Move diagonally down and left.

F [<n>] Move diagonally down and right.

### Other Commands

M <x,y> Move absolute or relative. If x is preceded by a plus (+) or minus (-), x and y are added to the current graphics position and connected with the current position by a line. If not, a line is drawn to point x,y from the current cursor position.

A <n> Set angle n. n may range from zero to three where zero is zero degrees, one is 90, two is 180, and three is 270. Figures rotated 90 or 270 degrees are scaled so they appear the same size as with zero or 180 degrees on a monitor screen with the standard aspect ratio of 4/3.

TA <degrees> - rotate <degrees>. DEGREES must be in the range - 360 to 360 degrees. If DEGREES is positive, rotation is anti-clockwise; if negative, rotation is clockwise.

### Example

```
FOR D=0 TO 360 'draw spokes.  
DRAW "TA=D;NU50"  
NEXT D
```

C <n> Set colour n.

S <n> Set scale factor. n may range from one to 255. The scale factor multiplied by the distances given with U, D, L, R, or relative M commands gives the actual distance travelled.

X <string expression> Execute substring. With this strong command each string can prompt another, much like GOSUB. Numeric arguments can be constants like "123" or "= <variable>" where <variable> is named.

P <paintcolour>, <bordercolour>. Paintcolour is an integer paint attribute and bordercolour the integer border attribute. You cannot "Tile Paint" with DRAW.

### Examples

```
DRAW "U50R50D50L50" 'Draw a box  
DRAW "BE10" 'Move up and right into box.  
DRAW "P1,3" 'Paint interior.
```

```
10 U$="U30;"  
20 D$="D30;"  
30 L$="L40;"  
40 R$="R40;"  
50 BOX$=U$+R$+D$+L$  
60 DRAW "XBOX$;"
```

The statement DRAW "XU\$;XR\$;XD\$;XL\$;" draws the same box.

## EDIT Command

Edits the specified line.

### Syntax

EDIT < line number >

When EDIT is used, GW-BASIC types the specified program line and stays in direct mode. The cursor is on the first character of the program line.

See section 3.

## END Statement

Terminates a program, closes all files, and returns to command level.

### Syntax

END

END statements can be used anywhere in the program. Unlike STOP a "Break in line nnnnn" message is not printed. END is optional.

### Example

```
520 IF K > 1000 THEN END ELSE GOTO 20
```

## ENVIRON Statement

Modify a parameter in MS-DOS's environment string table.

### Syntax

ENVIRON *stringexpression*

The *stringexpression* must be of the form *parameterid* = *text*, or *parameterid text*. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right, text.

If the *parameterid* has not previously existed in the environment string table, it is appended to the end of the table. If a *parameterid* exists on the table when the ENVIRON statement is executed, it is deleted and the new *parameterid* is appended to the end of the table.

The text string is the new parameter text. If the text is a null string (""), or a semicolon (";"), then the existing parameter-id is removed from the environment string table, and the remaining body of the file is compressed.

You can use this statement to change the PATH parameter for a child process, or to pass parameters to a child by inventing a new environment parameter.

Errors include parameters that are not strings and an "Out of memory" when no more space can be allocated to the environment string table. The amount of free space in the table will usually be quite small.

### Example

The following operating system command will create a default PATH to the root directory on disk A:

```
PATH = A:\
```

The PATH may be changed to a new value by:

```
ENVIRON "PATH = A:\SALES;A:\ACCOUNTING"
```

A new parameter may be added to the environment string table:

```
ENVIRON "SESAME = PLAN"
```

The environment string table now contains:

```
PATH = A:\SALES;A:\ACCOUNTING  
SESAME = PLAN
```

If you then entered:

```
ENVIRON "SESAME = ;"
```

you would have deleted SESAME, and you would have a table containing:

```
PATH = A:\SALES;A:\ACCOUNTING
```

## ENVIRON\$ Function

Retrieves an environment string from BASIC's environment String table.

### Syntax

ENVIRON\$ (*environstring*)  
ENVIRON\$ (*n*)

The argument *n* is an integer.

The string result returned by the ENVIRON\$ function may not exceed 255 characters. If you specify a *environstring* name, but it either cannot be found or does not have any text following it, then ENVIRON\$ returns a null string. If you specify a *environstring* name, ENVIRON\$ returns all the associated text that follows *environstring*= in the environment string table.

If the argument is numeric, then the *n*th string in the environment string table is returned. The string in such a case, includes all the text, including the *environstring* name. If the *n*th string does not exist, a null string is returned.

## EOF Function

Tests for the end-of-file condition.

### Syntax

EOF( <file number> )

Returns - 1 (true) if the end of a sequential file has been reached. Use while inputting, to avoid "Input past end" errors.

Used with random access files, it returns "true" if the last GET statement was unable to read an entire record through trying to read beyond the end.

When used with a communications device, the end-of-file condition depends on the opening mode (ASCII or binary). In binary mode, EOF is true when the input queue is empty ( $\text{LOC}(n) = 0$ ). It becomes false when the input queue is not empty. In ASCII mode, EOF is false until a CTRL-Z is received, and it then remains true until the device is closed.

### Example

```
10 OPEN "I",1,"DATA"  
20 C=0  
30 IF EOF(1) THEN 100  
40 INPUT #1,M(C)  
50 C=C+1:GOTO 30  
.  
.  
.
```

## ERASE Statement

Eliminates arrays from memory.

### Syntax

ERASE <list of array variables>

Arrays can be redimensioned after they are erased, or the memory space used for other purposes. Redimensioning an array without first erasing it causes a "Duplicate definition" error.

### Example

```
.  
.  
.  
450 ERASE A,B  
460 DIM B(99)  
.  
.  
.
```

## ERDEV, ERDEV\$ Functions

Provides a way to obtain device-specific status information.

### Syntax

ERDEV  
ERDEV\$

ERDEV is an integer function that contains the error code returned by the last device to declare an error. ERDEV\$ is a string function which contains the name of the device driver which generated the error.

You may not set these functions.

ERDEV is set by the interrupt 24 handler when an error within the operating system is detected.

ERDEV returns the DOS INT 24 error code in its lower eight bits. The upper eight bits returned contain the device attribute word. The bits of the device attribute word, are returned as the upper eight bits of the word returned by ERDEV in the following order: 15, 14, 13, XX, 3, 2, 1, 0. The XX value is always zero.

### Example

If a user-installed device driver, MYLPT2, runs out of paper, and the driver's error number for that problem is 9, then

```
PRINT ERDEV, ERDEV$
```

prints this output:

```
9           MYLPT2
```

## ERR and ERL FUNCTIONS

### Syntax

ERR ERL

When an error handling routine is entered, ERR contains the code for the error, ERL the line number of the detected error. The ERR and ERL functions are usually used in IF...THEN statements to direct program flow in the error handling routine.

If the statement causing the error was a direct mode type ERL contains 65535.

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. As ERL and ERR are reserved words, they can not appear to the left of the equal sign in a LET (assignment) statement. See the Appendix for error codes.

### Example

To test whether an error occurred in a direct statement enter:

```
IF 65535 = ERL THEN PRINT "Direct Error"
```

When testing within a program, use:

```
IF ERR = error code THEN ...  
IF ERL = line number THEN ...
```

## ERROR Statement

Simulates a BASIC error or defines error codes.

### Syntax

ERROR <integer expression>

ERROR can be used as a statement (part of a program source line) or as a command (in direct mode).

The value of integer expression must be greater than zero and less than 256. If its value equals an error code already in use by BASIC the ERROR statement simulates that error and the corresponding error message printed. (See Example 1.)

In defining an error code use the highest available values so compatibility may be maintained when more error codes are added.

If an ERROR statement specifies a code for which no error message has been defined an "Unprintable error" message results. If there isn't a handling routine for an error statement an error message is printed and the program halts.

### Example 1

```
20 S=15  
30 ERROR S  
40 END
```

returns

```
String too long in line 30
```

Or, in direct mode (interpreter only):

```
Ok  
ERROR 15  
String too long  
Ok
```

(ERROR 15 was your response; String too long was GW-BASIC's response.)

## Example 2

```
.  
. .  
110 ON ERROR GOTO 400  
120 INPUT "WHAT IS YOUR BET";B  
130 IF B > 5000 THEN ERROR 210  
. .  
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS  
$5000"  
410 IF ERL = 130 THEN RESUME 120  
. .
```

## EXP Function

Returns  $e$  (base of natural logarithms) to the power of  $X$ .  $X$  must be  $\leq 88.02969$ .

### Syntax

EXP( $X$ )

If ( $x$ ) is greater than 88.02969, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied and the program continues.

The EXP function returns a single precision value unless the /D switch was used when BASIC was invoked and a double precision variable is used as the argument.

### Example

```
10 X=5  
20 PRINT EXP(X-1)
```

returns

54.59815

## EXTERR Function

Returns extended error information

### Syntax

EXTERR (*n*)

EXTERR returns “extended” error information provided by versions of DOS 3.0 and greater. For versions of DOS earlier than 3.0, EXTERR always returns zero. The single integer argument must be in the range 0-3 as described in the table below.

#### EXTERR function return values

---

<i>n</i>	Return Value
0	Extended error code
1	Extended error class
2	Extended error suggested action
3	Extended error locus

---

The values returned are *not* defined by BASIC but by DOS. Refer to the DOS Programmer’s Reference (version 3.0 or later) for a description of the values returned by the DOS extended error function.

The extended error code is actually retrieved and saved by BASIC each time appropriate DOS functions are performed. Thus when an EXTERR function call is made, these saved values are returned.

## FIELD Statement

Allocates space for variables in a random file buffer.

### Syntax

FIELD [#] <file number>, <field width> AS <string variable>

<file number> is the number of the opened file;

<field width> is the number of characters allocated to the string variable.

FIELD must be used to format the random file buffer before a GET or PUT statement.

The number of bytes allocated in a FIELD statement must not exceed the length specified when the file was opened. Otherwise a "Field overflow" error occurs. The default record length is 128 bytes.

Any number of FIELD statements can be used for the same file. They remain in effect at the same time.

### Note

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. In any subsequent INPUT or LET statement with the same name that variable no longer refers to the random file record buffer, but to the variables stored in string space.

### Example 1

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

Allocates the first 20 bytes in the random file buffer to the string variable N\$, the next 10 bytes to ID\$, and the next 40 to ADD\$. FIELD does not place any data in the random file buffer. (See also GET, LSET and RSET statements).

## Example 2

```
10 OPEN "R","#1,"A:PHONELST",35
15 FIELD #1,2 AS RECNR$,33 AS DUMMY$
20 FIELD #1,25 AS NAMES,10 AS PHONENBR$
25 GET #1
30 TOTAL=CVI(RECNR)$
35 FOR I=2 TO TOTAL
40 GET #1,I
45 PRINT NAMES, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

## Example 3

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS
  A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration: FIELD #1,16 AS A\$(0),16 AS A\$(1),...,16 AS A\$(6),16 AS A\$(7)

#### Example 4

```
10 DIM SIZE% (4%): REM ARRAY OF FIELD SIZES
20 FOR LOOP%= 0 TO 4%
30 READ SIZE% (LOOP%)
40 NEXT LOOP%
50 DATA 9,10,12,21,41
.
.
.
120 DIM A$(4%): REM ARRAY OF FIELDDED
VARIABLES
130 OFFSET%= 0
140 FOR LOOP%= 0 TO 4%
150 FIELD #1,OFFSET% AS OFFSET$,SIZE%
(LLOOP%) AS A$(LOOP%)
160 OFFSET%=OFFSET% + SIZE%(LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3.  
However, the element size varies. The equivalent  
declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4%) AS A$(4%)
```

## FILES Statement

Prints the names of files on the specified disk.

### Syntax

FILES [ < filespec > ]

< filespec > includes a filename or a pathname and optional device designation. It is a string formula which may contain question marks (?) or asterisks (\*) used as wild cards.

A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at that position. It is shorthand for a series of question marks. The asterisk need not be used where all the files on a drive are requested, for example, FILES "B:".

If a filespec is used, and no explicit path is given, the current directory is the default. If omitted, all the files on the currently selected drive are listed.

### Examples

```
FILES
```

Shows all files on the current directory.

```
FILES "*.BAS"
```

Shows all files with extension .BAS.

```
FILES "B:*.*)"
```

Shows all files on drive B.

```
FILES "B:" (equivalent to "B:*.*)"
FILES "TEST?.BAS"
```

Shows all five-letter files whose names start with "TEST" and end with the .BAS extension.

```
FILES "\SALES"
```

If SALES is a subdirectory of the current directory, this statement displays SALES <dir> . If SALES is a file in the current directory, this statement displays SALES.

```
FILES "\SALES\MARY"
```

Displays MARY <dir> if MARY is a subdirectory of SALES or if MARY is a file, displays its name.

## FIX Function

Returns the truncated integer part of X.

### Syntax

FIX(X)

FIX(X) is equivalent to  $\text{SGN}(X) * \text{INT}(\text{ABS}(X))$ . It differs from INT in that it does not return the next lower number for negative X.

### Examples

```
PRINT FIX(58.75)
```

returns

```
58
```

```
PRINT FIX(-58.75)
```

returns

```
- 58
```

## FOR...NEXT Statement

Allows a series of instructions to be performed in a loop a given number of times.

### Syntax

```
FOR <variable> = x TO y [STEP z]
```

```
.
```

```
NEXT [<variable> ][,<variable> ...]
```

x, y, and z are numeric expressions.

Variable is used as a counter. The first numeric expression (x) is the initial value. The second numeric expression (y) is the final value. Following the FOR statement the program continues until the NEXT statement is met.

The counter is then adjusted by the amount specified by STEP. If the value of the counter is now greater than the final value (y) the program continues with the statement following NEXT. If the value is less GW-BASIC branches back to the statement after the FOR statement and the process is repeated. This is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop which continues until the counter is less than the final value.

The counter must be an integer or single precision numeric constant. If a double precision numeric constant is used, a "Type mismatch" error results.

The body of the loop is skipped if the initial value of the loop times the sign of the STEP exceeds the final value times the sign of the STEP.

### Nested Loops

FOR...NEXT loops can be nested by being placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement can be used for all of them.

The variable(s) in the NEXT statement can be omitted. Then the NEXT statement will match the most recent FOR statement. If a NEXT statement is met before its corresponding FOR statement, a "NEXT without FOR" error message is issued and the program ends.

### Example 1

```
10 K=10
20 FOR I = 1 TO 10 STEP 2
30 PRINT I;
40 LET K = K+10
50 PRINT K
60 NEXT I
```

returns

```
1 20
3 30
5 40
7 50
9 60
```

The loop counter, I, advances + 2 on each cycle. The loop prints the counter, increments K, and prints K.

### Example 2

```
10 J=0 20 FOR I= 1 TO J 30 PRINT I 40 NEXT I
```

The loop does not work because its initial value exceeds the final value.

### Example 3

```
10 I=5  
20 FOR I=1 TO I+5  
30 PRINT I;  
40 NEXT I
```

returns

```
1 2 3 4 5 6 7 8 9 10
```

The loop repeats ten times. The final value for the loop variable is always set before that of the initial value.

## FRE Function

With a numeric argument, FRE returns the number of bytes in memory not being used. Arguments to FRE are dummy ones.

### Syntax

```
FRE(0)  
FRE(“”)
```

FRE(“”) forces a garbage collection before returning the number of free bytes. Garbage collection is not initiated until all free memory has been used up. Use FRE(“”) periodically for shorter delays in this action.

### Example

```
PRINT FRE(0)
```

might return

```
14542
```

## GET Statement - File I/O

Reads a record from a random disk file into a random buffer.

### Syntax

GET [#] <file number> [, <record number> ]

<file number> is the number of the OPENed file. If the record number is omitted, the record after the last GET is read into the buffer. The largest possible record number is 16,777,215.

GET and PUT statements allow fixed-length input and output for GW-BASIC COM files. Do not use for telephone communication because of the low performance associated with telephone line communications.

### Example

```
GET #1,75
```

### Note

After a GET statement use INPUT# and LINE INPUT# to read characters from the random file buffer. Use EOF after a GET statement to check if the GET was beyond the end of file marker.

## GET Statement - Graphics

GET and PUT are used together to transfer graphic images to and from the screen.

### Syntax

GET (x1,y1)-(x2,y2), <array name>

used with

PUT (x1,y1), <array name> [,action verb]

(x1,y1) - (x2,y2) is a rectangular area on the display screen. The rectangle is defined with (x1,y1) and (x2,y2) being the upper-left and the lower-right vertices.

The array name is the placename holding the image. It can be any type except string and must be dimensioned large enough to hold the entire image. If not type integer the contents of the array after a GET will be meaningless when interpreted directly.

The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The PUT statement transfers the image stored in the array onto the screen.

GET and PUT are extremely useful for animation. See PUT for more information.

## GOSUB...RETURN Statements

Branches to and returns from a subroutine.

### Syntax

GOSUB <line number>

·  
·  
·

RETURN [<line number>]

<line number> is the first line of the subroutine. It can be called any number of times in a program and may also be called from another subroutine. Nesting of subroutines is limited only by available memory.

A subroutine can contain more than one RETURN statement. After a simple RETURN statement in a subroutine GW-BASIC returns to the statement following the most recent GOSUB.

You can also use the line number option in RETURN can be used to return to a specific line number. Use with care as any other GOSUBs, WHILEs, or FORs active at the time of the GOSUB remain so, and errors such as "FOR without NEXT" can result.

Subroutines may appear anywhere in the program, but they should be easy to distinguish from the main program. To prevent accidental entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

### Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT "IN";
60 PRINT "PROGRESS"
70 RETURN
```

returns

```
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
```

## GOTO Statement

Branches unconditionally to a specified line number.

### Syntax

GOTO <line number>

<line number> defines where the program begins; that is, at that line or at the first executable statement after that.

### Example

```
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
```

returns

```
R = 5      AREA = 78.5  
R = 7      AREA = 153.86  
R = 12     AREA = 452.16  
Out of DATA in 10
```

## HEX\$ Function

Returns a string representing the value of the decimal argument.

### Syntax

HEX\$(X)

X is rounded to an integer before HEX\$(X) is evaluated.

### Example

```
10 INPUT X  
20 A$ = HEX$(X)  
30 PRINT X "DECIMAL IS" A$ "HEXADECIMAL"
```

returns

```
? 32  
32 DECIMAL IS 20 HEXADECIMAL
```

See the OCT\$ function for octal conversion.

## IF..THEN[...ELSE]/IF...GOTO Statements

Makes a decision on program flow based on the result returned by an expression.

### Syntax

```
IF <expression> [,] THEN { <statement(s)> :  
<line number> }
```

```
[, [ELSE { <statement(s)> :<line number> }]]
```

```
IF <expression> [,] GOTO <line number>
```

```
[, [ELSE { <statement(s)> :<line number> }]]
```

If the result of the expression is not zero, the THEN or GOTO clause is put into effect. THEN may be followed by a line number for branching or any statements to be made. GOTO is always followed by a line number. If the result of the expression is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, used. The program continues with the next executable statement. A comma is allowed before THEN.

IF...THEN...ELSE statements can be nested. Line length is the only limit.

### Example

```
IF X>Y THEN PRINT "GREATER" ELSE IF X>Y  
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN.

### Example

```
IF A = B THEN IF B = C THEN PRINT "A = C" ELSE  
PRINT "A < > C"
```

will not print "A < > C" when A < > B.

Unless a statement with the specified line number has been entered in indirect mode an IF...THEN statement followed by a line number in direct mode results in an "Undefined line" error.

## Note

When using IF to test equality for a value that is the result of a floating-point computation, the internal representation of the value may not be exact. The test should be against the range over which the accuracy of the value may vary.

## Example

To test a computed variable A against the value 1.0 use;

```
IF ABS (A-1.0) < 1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

## Example 1

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if one is not zero.

## Example 2

```
100IF(I < 20)*(I > 10) THEN DB = 1979-1: GOTO 300  
110 PRINT "OUT OF RANGE"
```

Here a test determines if I is greater than 10 and less than 20. If one is in this range, DB is calculated and execution branches to line 300. If I is not in this range the program continues with line 110.

## Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

Printed output goes to the screen or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise it goes to the screen.

## INKEY\$ Function

Returns either a one-character string containing a character read from the standard device or a null string if there isn't a character pending. The keyboard is usually the standard input device.

### Syntax

INKEY\$

No characters are echoed. All are passed through to the program except for CTRL-C which ends the program.

### Example

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$ = ""
1020 FOR I% = 1 TO TIMELIMIT%
1030 A$ = INKEY$
1035 IF LEN(A$) = 0 THEN 1060
1040 IF ASC(A$) = 13 THEN TIMEOUT% = 0
1045 IF TIMEOUT% = 0 THEN RETURN
1050 RESPONSE$ = RESPONSE$ + A$
1060 NEXT I%
1070 TIMEOUT% = 1 : RETURN
```

### Note

Some keys may return a two-byte string. This depends on the machine used.

## INP Function

Returns the byte read from port I. I must be in the range zero to 65535.

### Syntax

INP(I)

INP complements the OUT statement.

### Example

```
100 A = INP(54321)
```

In 8086 assembly language, this is equivalent to  
MOV DX, 54321  
IN AL, DX

## INPUT Statement

Allows input from the keyboard during a program.

### Syntax

INPUT[;] [<"prompt string">;] <list of variables>

When an INPUT statement is met, the program pauses and a question mark printed to indicate it is waiting for data. If "prompt string" is included, the string is printed before the question mark. Data is then entered at the keyboard.

BASIC can be redirected to read from standard input and write to standard output by providing the input and output filenames at the start.

A comma can be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE", B\$ will print the prompt without a question mark.

If INPUT is followed by a semicolon, the carriage return typed to input data does not echo a carriage return/linefeed sequence.

The data entered is assigned to the variables given in the list. The number of data items supplied and the number of variables in the list must be the same. Data items are separated by commas.

The variable names may be numeric or string and include subscripted variables. The type of each data item input must agree with the type specified by the variable name. Strings input to an INPUT statement need not be surrounded by quotation marks.

If the wrong response to INPUT is made a "?Redo from start" is printed. Input values are not assigned until an acceptable response is given.

### Examples

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
```

returns

```
? 5
5 SQUARED IS 25
```

You entered 5 in response to the question mark.

```
10 PI = 3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A = PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
```

returns

```
WHAT IS THE RADIUS? 7.4
```

You entered 7.4 in response to the question mark.

```
THE AREA OF THE CIRCLE IS 171.946
WHAT IS THE RADIUS?
```

and so on.

## INPUT# Statement

Reads data items from a sequential device or file and assigns them to program variables.

### Syntax

`INPUT#<file number> , <variable list>`

<file number> is the number of the file OPENed for input; <variable list> contains the names to be assigned to the items in the file. The variable type must match the type specified by the variable name. With INPUT#, no question mark is printed.

The data items in the file should appear like data entered in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and linefeeds are ignored. The first character encountered that isn't a space, carriage return, or linefeed is assumed to be the start of a number. The number ends on a space, carriage return, linefeed, or comma.

This also applies to sequential data being scanned for string items. If the first character met is a quotation mark (""), the string item will consist of all characters read between the first quotation mark and the second. A quoted string can't therefore contain a quotation mark as a character.

If the first character of the string isn't a quotation mark, the string is unquoted and will end on a comma, carriage return, or linefeed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

### Example

```
INPUT 2,A,B,C
```

## INPUT\$ Function

Returns a string of X characters, reads from file number Y. If the file number is not specified, the characters are read from the standard input device.

### Syntax

INPUT\$(X[, [#]Y])

If the keyboard is used for input, characters are not echoed on the screen. All control characters are passed through except Break which interrupts the INPUT\$ function.

The keyboard is the standard input device. However, BASIC can be re-directed to read from standard input by providing the input filename on the command line. See Section 4.4.

### Example 1

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
  HEXADECIMAL
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

### Example 2

```
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

## INSTR Function

Searches for the first occurrence of string Y\$ in X\$, and returns the matched position.

Optional offset I sets the position for starting the search.

### Syntax

INSTR([I,]X\$,Y\$)

I must be in the range one to 255. If I is greater than the number of characters in X\$ (LEN(X\$)), or if X\$ is null or Y\$ cannot be found, INSTR returns zero.

If Y\$ is null, INSTR returns I or one, and if no I was specified, then INSTR returns one. X\$ and Y\$ may be string variables, string expressions, or string literals.

### Example

```
10 X$="ABCDEB"  
20 Y$="B"  
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
```

returns

2

6

## INT Function

Returns the largest integer  $\leq X$ .

### Syntax

INT(X)

### Examples

```
PRINT INT(99.89)
```

returns

```
99
```

```
PRINT INT(-12.11)
```

returns

```
-13
```

See the CINT and FIX functions which also return integer values.

## IOCTL Statement

Transmits a control character or string to a device driver.

### Syntax

IOCTL [#]*filename*, *string*.

IOCTL commands are generally two to three characters followed optionally by an alphanumeric argument. An IOCTL\$ command string may be up to 255 bytes long.

The IOCTL statement works only if:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. BASIC performs an OPEN on a file on that device.

Most standard MS-DOS device drivers don't process IOCTL strings, and it is necessary for you to determine if the specific driver can handle the command. See also IOCTL\$.

### Example

If you wanted to set the page length to 66 lines per page on LPT1, your procedure might look like this:

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS 1
20 IOCTL$ 1, "PL66"
```

## IOCTL\$ Function

Receives a control data string from a device driver.

### Syntax

IOCTL\$ ([#]*filenumber*)

The IOCTL\$ function is most frequently used to receive acknowledgement that an IOCTL statement succeeded or failed, or to obtain current status information.

You could use IOCTL\$ to ask a communications device to return the current baud rate, information on the last error, logical line width, and so on.

The IOCTL\$ function works only if:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. BASIC performs an OPEN on a file on that device.

See also IOCTL.

### Example

This example tells the device that the data is raw:

```
10 OPEN "\DEV\ENGINE" AS 1
20 IOCTL #1, "RAW"
```

In this continuation, if the character driver ENGINE responds "false" from the raw data mode IOCTL statement, then the file is closed:

```
30 IF IOCTL$ (1) = "0" THEN CLOSE 1
```

## KEY STATEMENT

Assigns softkey values to function keys and displays the values.

### Syntax

KEY n, X\$  
KEY LIST  
KEY ON  
KEY OFF

(n) is the number of the function key. X\$ is the text assigned to the specified key.

Each of the function keys can be assigned a 15-byte string which is input to GW-BASIC when that key is pressed.

KEY ON, KEY OFF, and KEY LIST statements display the softkeys.

KEY ON displays the softkey values on the bottom line of the screen.

KEY OFF erases the softkey display from the bottom line, making it available for program use. It does not disable the function keys.

KEY LIST displays all softkey values on the screen, with all 15 characters of each key displayed.

Assigning a null string (string of length zero) to a softkey disables the function key as a softkey.

If the function key number is not in the range of permissible function key numbers, an "Illegal function call" error results, and the previous key expression is retained.

When a softkey is assigned, the INKEY\$ function returns one character of the softkey string per invocation.

### Example

```
50 KEY ON 'Displays the softkey on bottom line.  
60 KEY OFF 'Erases softkey display.  
70 KEY 1, "MENU" + CHR$(13)
```

Assigns the string "MENU" followed by a carriage return to softkey 1.

Use such assignments to speed data entry.

```
80 KEY 1," " 'Disables softkey 1.
```

The following routine initialises the first five softkeys:

```
10 KEY OFF 'Turns off key display during initialisation
20 DATA "EDIT","LET","SYSTEM","PRINT",
  "LPRINT"
30 FOR I = 1 TO 5
40 READ SOFTKEYS$(I)
50 KEY I,SOFTKEYS$(I)
60 NEXT I
70 KEY ON 'Displays new softkeys.
```

## KEY(n) Statement

Enables or disables event trapping of softkey or cursor direction key activity for the specified trappable key.

### Syntax

```
KEY(n) ON
KEY(n) OFF
KEY(n) STOP
```

(n) is the number of a function, a user-defined or a cursor direction key.

The function keys are followed by the cursor direction keys in the following order, up, left, right, down.

Then come the six user defined keys. They are defined by the statement:

```
KEY i,CHR$(j) + CHR$(k)
```

I is the user key number. J and K are characters which uniquely define a key on the keyboard.

Note that the KEY statement previously described assigns softkey and cursor direction values to function keys and displays the values. Do not confuse KEY ON and KEY OFF, which display and erase these values, with the event trapping statements described in this section.

The KEY(n) ON statement enables softkey or cursor direction key event trapping by an ON KEY statement (see ON KEY Statement.) While trapping is enabled, and if a non-zero line number is specified in the ON KEY statement, GW-BASIC checks between every statement to see if a softkey or cursor direction key has been used. If it has, the ON KEY statement is carried out. The text normally associated with a function key is not printed.

KEY(n) OFF disables the event trap. If an event takes place, it is not remembered.

KEY(n) STOP disables the event trap, but if an event occurs, it is remembered and an ON KEY statement carried out as soon as trapping is enabled.

### Note

For additional information on key event trapping, see section 2.7 and the ON KEY Statement.

### Example

```
10 KEY 4,SCREEN 0,0 'assigns softkey 4.  
20 KEY(4) ON 'enables event trapping.
```

```
.  
. .  
. .
```

```
70 ON KEY(4) GOSUB 200
```

```
.  
. .  
. .
```

key 4 pressed

```
.  
. .  
. .
```

```
200 'subroutine for the screen.
```

## KILL Statement

Deletes a file or a pathname from disk.

### Syntax

KILL [ <filespec> ]

If a KILL statement is given for a file already open, a "File already open" error occurs.

KILL is used for all types of disk files: program, random data and sequential data files. The filespec may contain question marks (?) or asterisks (\*) used as wildcards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at its position.

Since it is possible to reference the same file in a sub-directory via different paths, it is nearly impossible for BASIC to know that it is the same file simply by looking at the path.

For example; if MARY is your current directory, then:

```
"REPORT" ...  
"\SALES\MARY\REPORT" ...  
".. \MARY\REPORT" ...  
"... \MARY\REPORT" ...
```

all refer to the same file. Any open file with the same file name causes a "file already open" error.

### Warning

Be extremely careful when using wildcards with this command.

### Examples

```
200 KILL "DATA1?.DAT"
```

The position of the question mark will match any valid filename character. This command kills any file having a six character name starting with "DATA1" and having the filename extension ".DAT". This includes "DATA10.DAT" and "DATA1Z.DAT".

## Examples

```
210 KILL "DATA1.*"
```

Kills all files named DATA1, regardless of the filename extension.

```
220 KILL "..\GREG\*.DAT"
```

Kills all files with the extension ".DAT" in a directory called GREG.

## LEFT\$ Function

Returns a string comprising the leftmost I characters of X\$.

### Syntax

LEFT\$( <string> ,I)

I must be in the range zero to 255. If I is greater than the number of characters in string, (LEN(X\$)), the entire string is returned. If I = 0, the null string (length zero) is returned.

### Example

```
10 A$="BASIC LANGUAGE"  
20 B$=LEFT$(A$,5)  
30 PRINT B$
```

returns

```
BASIC
```

See also the MID\$ and RIGHT\$ functions.

## LEN Function

Returns the number of characters in <string> . Nonprinting characters and blanks are counted.

### Syntax

LEN( <string> )

### Example

```
10 X$="BIRMINGHAM, WEST MIDLANDS"  
20 PRINT LEN(X$)
```

returns

```
25
```

## LET Statement

Assigns the value of an expression to a variable.

### Syntax

[LET] <variable> = <expression>

LET is optional; the equal sign is sufficient for assigning an expression to a variable name.

### Example

```
110 LET D=12  
120 LET E=12^2  
130 LET F=12^4  
140 LET SUM=D+E+F  
.  
.  
.
```

or

```
110 D=12  
120 E=12^2  
130 F=12^4  
140 SUM=D+E+F  
.  
.  
.
```

## LINE Statement

Draws a line or box on the screen.

### Syntax

```
LINE [[STEP](x1,y1)]-[STEP](x2,y2)  
[, [<colour>] [,b[f]]] [,style]
```

(x1,y1) is the coordinate for the starting point of the line.

(x2,y2) is the ending point for the line.

<colour> is the colour number of the line to be drawn. (See COLOUR statement). If the ,b or ,bf option is used, the box is drawn in this colour.

,b draws a box with the points (x1,y1) and (x2,y2) specifying the upper left and lower right corners.

,bf draws a filled box.

,style is a 16-bit integer mask used when putting pixels down on the screen. This is called "line styling".

When coordinates specify a point not in the current viewport, the line segment is clipped to the viewport.

The relative coordinate form STEP (xoffset,yoffset) can be used in place of an absolute coordinate. For example, assume that the most recent point referenced was (10,10). The statement LINE STEP (10,5) would specify a point at offset 10 from x and offset 5 from y, that is, (20,15).

If the STEP option is used for the second coordinate on a LINE statement, it is relative to the first coordinate in the statement. Other ways to establish a new "most recent point" are through initialising the screen with the CLS and SCREEN statements. PSET, PRESET, CIRCLE and DRAW also establish a new "most recent point".

Each time LINE stores a point on the screen it uses the current circulating bit in style. If this is zero, then no storing is done; if the bit is a one the point is stored. After each point is stored, the next bit position in style is selected. Since there is no change to the point on screen with a zero bit in style a background line can be drawn before a "styled" line to force a known background. Style is used for normal lines and boxes, but has no effect on filled boxes.

## Examples

The following examples assume a screen 320 pixels wide by 200 pixels high.

```
10 LINE - (x2,y2)
```

Draws a line from the last point to x2,y2 in the foreground colour.

```
20 LINE (0,0) - (319,199)
```

Draws a diagonal line across the screen (downward).

```
30 LINE (0,100) - (319,100)
```

Draws a line across the screen.

```
40 LINE (10,10) - (20,20),2
```

Draws a line in colour 2.

```
10 FOR x = 0 to 319  
20 LINE (x,0) - (x,199),x AND 1  
30 NEXT
```

Draws an alternating line on-line off pattern on a monochrome display.

```
10 LINE (0,0) - (100,100),,b
```

Draws a box in the foreground (note that the colour is not included).

```
20 LINE STEP (0,0) - STEP (200,200),2,bf
```

Draws a filled box in colour 2.  
Coordinates are given as offsets.

```
10 LINE (0,0) - (160,100),3,,&HFF00
```

Draws a dashed line from the upper left hand corner to the centre of the screen.

## LINE INPUT Statement

Inputs an entire line of upto 254 characters to a string variable, without the use of delimiters.

### Syntax

LINE INPUT[:] [ <"prompt string">;]  
<string variable>

<"prompt string"> is a string literal printed at the terminal before input is accepted.

A question mark is not printed unless it is part of "prompt string". All input from the end of "prompt string" to the carriage return is assigned to the string variable. If a linefeed/ carriage return sequence is encountered both characters are echoed; but the carriage return is ignored, the linefeed put into the string variable and data input continues.

If LINE INPUT is followed by a semicolon, the carriage return typed to end the input line does not echo a carriage return/linefeed sequence at the terminal.

A LINE INPUT statement may be aborted by using CTRL-C. GW-BASIC returns to command level. With the Interpreter and through using CONT the program resumes at the LINE INPUT.

### Example

See LINE INPUT# Statement.

## LINE INPUT# Statement

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

### Syntax

LINE INPUT# <file number> , <string variable>

<file number> is the number of the OPENed file; <string variable> is the variable name to which the line is assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/linefeed sequence. The next LINE INPUT# reads all characters up to the next carriage return. A linefeed/carriage return sequence encountered is preserved.

LINE INPUT# is useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program. (See SAVE Command.)

When GW-BASIC is invoked with redirected input and output, all LINE INPUT statements read the input file specified instead of the keyboard.

GW-BASIC reads from this source until a CTRL-Z is detected. This condition can be tested with the EOF function. If the file is not ended by CTRL-Z or a BASIC file input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and BASIC returns to MS-DOS.

### Example

```
10 OPEN "O",1,"LIST"  
20 LINE INPUT "CUSTOMER INFORMATION?" ;C$  
30 PRINT #1, C$  
40 CLOSE 1  
50 OPEN "I",1,"LIST"  
60 LINE INPUT #1, C$  
70 PRINT C$  
80 CLOSE 1
```

returns

```
CUSTOMER INFORMATION? LINDA JONES 234, WIGAN  
LINDA JONES 234, WIGAN
```

## LIST Command

Lists all or part of the program currently in memory.

### Syntax

```
LIST [ <line number> ] [- [ <line number> ] ]  
[, <device> ]
```

<line number> is in the range zero to 65529; <device> is a designation string, such as LPT: or a filename.

If line number is omitted, the program is listed from the lowest line number. Listing ends when the end of the program is reached or when Break is used. When line number is included, only the specified line is listed.

If only the first line number is specified, that line and all higher-numbered lines are listed; with only the second line number specified, all lines from the beginning of the program through that line are listed.

When both line numbers are specified, the entire range is listed.

If the device is omitted, the listing is shown at the terminal.

GW-BASIC returns to command level after a LIST.

### Examples

```
LIST
```

Lists the program currently in memory.

```
LIST 500
```

Lists line 500.

```
LIST 150-
```

Lists all lines from 150 to the end.

```
LIST -1000
```

Lists all lines from the lowest number through 1000.

```
LIST 150-1000
```

Lists lines 150 through 1000, inclusive.

```
LIST 150-1000,"LPT:"
```

Lists lines 150 through 1000 on the line printer.

## LLIST Command

Lists all or part of the program currently in memory on the line printer.

### Syntax

LLIST [<line number> [- [<line number> ]]]

LLIST assumes a 132-character-wide printer.

The options are the same as for LIST Command and LLIST works in a similar way to the examples shown for the LIST Command (except for the last one addressing a device).

GW-BASIC returns to command level after a LLIST.

## LOAD Command

Loads a file from an input device into memory.

### Syntax

LOAD <filespec> [,R]

For loading a program, the filespec is an optional device specification followed by a conforming filename. The filespec must include the filename used when the file was saved or created. BASIC adds the default filename extension .BAS if no extensions are specified when the file is saved to the disk.

LOAD closes all open files and deletes all variables and program lines currently in memory before loading the designated program. If the R option is used with LOAD, the program runs after it is loaded, and all open data files are kept open. Use this method to chain several programs, or segments of the same program. Information may be passed between the programs using their disk data files.

### Example

```
LOAD "STRTRK",R
```

Loads and runs the program STRTRK.BAS

```
LOAD "B:MYPROG"
```

Loads the program MYPROG.BAS from the disk in drive B, but does not run the program.

## LOC Function

With random disk files, LOC returns the actual record number within the file. With sequential files, LOC returns the current byte position in the file, divided by 128.

### Syntax

LOC(<file number> )

<file number> is the number of the opened file.

For a communications file, LOC(X) sees if there are any characters in the input queue waiting to be read. If there are more than 255 characters waiting, LOC(X) returns 255. Since interpreter strings are limited to 255 characters, there is no need to test for string size before reading data into it.

If fewer than 255 characters remain in the queue, the value returned by LOC(X) depends on whether the device was opened in ASCII or binary mode. In either mode, LOC returns the number of characters that can be read from the device. In ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. Any attempt to do so results in an "Input past end" error.

### Example

```
200 IF LOC(1) > 50 THEN STOP
```

## LOCATE Statement

Moves the cursor to the specified position. Optional parameters turn the blinking cursor on and off and define the vertical start and stop lines.

### Syntax

```
LOCATE [row][,[col]][,[cursor]][,[start][,stop]]]
```

<row> is a vertical line number on the screen. It should be a numeric expression returning an unsigned integer.

<col> is the column number on the screen. It should be a numeric expression returning an unsigned integer.

<cursor> is a Boolean value indicating whether or not the cursor should be visible .

<start> is the cursor vertical starting line on the screen. It should be a numeric expression returning an unsigned integer.

<stop> is the cursor vertical stop line on the screen. It should be a numeric expression returning an unsigned integer.

Any value outside the specified ranges results in an "Illegal function call" error. Previous values are then retained.

Parameters can be omitted from the statement. If so, the previous value is assumed.

The start and stop lines are the raster lines specifying which pixels on the screen are lit. A wider range between the start and stop lines produces a cursor similar to one occupying an entire character block. The start and stop lines assume the same value if the latter is omitted.

The last line on the screen is reserved for softkey display. It is not accessible unless the softkey display is off and LOCATE used to move the cursor there.

### Example

```
10 LOCATE 1,1
```

Moves cursor to upper-left corner of the screen.

```
20 LOCATE ,,1
```

Makes the cursor visible; position remains unchanged.

30 LOCATE ...,7

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character starting and ending on raster line 7.

40 LOCATE 5,1,1,0,7

Moves the cursor to line 5, column 1; turns cursor on. Cursor covers entire character cell starting at scan line 0 and ending on scan line 7.

## LOCK...UNLOCK Statements

Controls access by other processes to all or part of another file.

### Syntax

LOCK [(#)]n[,[( <record number> )][TO <record number> )]

UNLOCK[(#)]n[,[( <record number> )][TO <record number> )]

These statements are supported only by the compiler. They are used in networked environments where several users might be working simultaneously on the same file. *n* is the number with which the file was opened.

If you specify just one record (record number), then only that record is locked or unlocked. If you specify a range of records and omit a starting record, then all records from the first record to the end of the range are locked or unlocked. LOCK with no record arguments locks the entire file, while UNLOCK with no record arguments unlocks the entire file.

If the file has been opened for *random* input or output, then the range indicates which records are to be locked or unlocked.

However, if the file has been opened for *sequential* input or output, LOCK and UNLOCK affect the entire file, regardless of the range specified.

### **Note**

Be sure to remove all locks with an UNLOCK statement before closing a file or terminating your program. Otherwise undefined results will occur.

LOCK and UNLOCK must match exactly.

### **Example 1**

The following locks the entire file opened as number 2:

```
lock #2 ]
```

The following locks only record 32 in file number 2:

```
lock #2, 32
```

The following locks records 1 to 32 in file number 2:

```
lock #2, to 32
```

### **Example 2**

The following UNLOCK would be legal:

```
lock #1, 1 to 4  
lock #1, 5 to 8  
unlock #1, 1 to 4  
unlock #1, 5 to 8
```

The following UNLOCK would be illegal, since the range in an UNLOCK statement must match the range in the corresponding LOCK statement exactly:

```
lock #1, 1 to 4  
lock #1, 5 to 8  
unlock #1, 1 to 8
```

Possible error messages generated by the LOCK statement include:

Permission denied - generated when syntactically correct LOCK request cannot be granted.

Illegal function call - generated when record range specified does not meet the necessary criteria, or when a range/record length combination exceeds the legal limit for the size of a file.

See UNLOCK statement below.

## LOF Function

Returns the length of the named file in bytes.  
when a file is opened for APPEND or OUTPUT.

### Syntax

LOF( <file number> )

### Example

```
110 IF REC*RECSIZ> LOF(1) THEN PRINT  
"INVALID ENTRY"
```

The variables REC and RECSIZ contain the record number and record length respectively. The calculation determines whether the specified record is beyond the end-of-file.

## LOG Function

Returns the natural logarithm of X. X must be greater than zero.

### Syntax

LOG(X)

### Example

```
PRINT LOG(45/7)
```

returns

```
1.860752
```

## LPOS Function

Returns the current position of the print head within the printer buffer.

### Syntax

LPOS(X)

(X) is the index of the printer being tested ; LPT1: is tested with LPOS(1), LPT2: with LPOS(2), and so on.

LPOS does not necessarily give the physical position of the print head.

### Example

```
100 IF LPOS(X) > 60 THEN LPRINT CHR$(13)
```

## LPRINT And LPRINT USING Statements

Prints data on the printer.

### Syntax

LPRINT [ <list of expressions> ]

LPRINT USING <string exp>; <list of expressions>

Same as PRINT and PRINT USING, except output goes to the line printer, and the file number option is not available.

LPRINT assumes a 132-character-wide printer but the width may vary according to machine.

## LSET And RSET Statements

Moves data from memory to a random file buffer in preparation for a PUT statement. Also left- or right-justifies the value of a string into a string variable.

### Syntax

LSET <string variable> = <string expression>

RSET <string variable> = <string expression>

If the string expression needs fewer bytes than were fielded to the string variable, LSET left-justifies the string in the field, and RSET right-justifies the string. Spaces are used to pad the extra positions. If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See MKI\$, MKS\$, MKD\$ functions.

### Examples

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = MKI$(COUNT%)
```

### Note

LSET or RSET can also be used with a nonfielded string variable to left-justify or right-justify a string in a given field.

### Example

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

This right-justifies the string N\$ in a 20-character field. Use for formatting printed output.

## MERGE Command

Merges a specified file into the program currently in memory.

### Syntax

MERGE <filespec>

For merging a program not in memory, the filespec is an optional device specification followed by a conforming filename or pathname. BASIC adds the default filename extension ".BAS" if no extensions are specified and the file has been saved to the disk. GW-BASIC returns to command level after a MERGE command.

Lines in disk files replace any lines in the program in memory bearing the same number.

### Example

```
MERGE "NUMBRS"
```

Inserts, by sequential line number, all lines in the program NUMBRS.BAS into the program currently in memory.

## MID\$ Statement

Replaces a portion of one string with another string.

### Syntax

MID\$( <string 1> ,n[,m]) = <string 2>

n and m are integer expressions .

The characters in string 1, beginning at position n, are replaced by the characters in string 2. The optional m refers to the number of characters from string 2 to be used in the replacement. If m is omitted, all of string 2 is used. Character replacement always ends at the original length of string 1.

## Example

```
10 A$="BIRMINGHAM,"
20 MID$(A$,12)="B15"
30 PRINT A$
```

returns

```
BIRMINGHAM, B15
```

MID\$ also returns a substring of a given string.

## MID\$ Function

Returns a string of length m characters from X\$, beginning with the nth character.

### Syntax

MID\$( <string> ,n[,m])

n and m must be in the range one to 255. If m is omitted or if there are fewer than m characters to the right of the nth character, all rightmost characters beginning with the nth character are returned. If n is greater than the number of characters in the string, that is, (LEN(<string>)), MID\$ returns a null string.

## Example

```
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
```

returns

```
GOOD EVENING
```

Also see the LEFT\$ and RIGHT\$ functions.

## MKDIR Statement

Creates a new directory.

### Syntax

MKDIR <pathname>

<pathname> is a string expression specifying the name of the directory to be created. MKDIR works like the MS-DOS command MKDIR. It must be less than 128 characters. See section 4.3 for tree-structured directories.

### Example

Assume the current directory is the root.

```
MKDIR "SALES"
```

Creates a sub-directory named SALES in the current directory of the current drive.

```
MKDIR "B:USERS"
```

Creates a sub-directory named USERS in the current directory of drive B.

Also see the CHDIR and RMDIR statements.

## MKI\$, MKS\$, MKD\$ Functions

Converts numeric values to string values.

### Syntax

MKI\$( <integer expression> )

MKS\$( <single precision expression> )

MKD\$( <double precision expression> )

Any numeric value placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string. MKS\$ converts a single precision number to a 4-byte string. MKD\$ converts a double precision number to an 8-byte string.

## Example

```
90 AMT = (K + T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

See also CVI, CVS, CVD Functions.

## NAME Statement

Changes the name of an existing disk file.

### Syntax

NAME <old filename> AS <new filename>

The old filename must be closed before the remaining command is carried out. The new filename must not exist or an error results.

Both files should be on the same drive on which there must be a one free handle. Any attempt to rename with a new drive designation leads to a "Rename across disks error". After a NAME command, the file exists on the same disk with the new name.

Do not use NAME to rename directories.

### Examples

```
NAME "ACCTS" AS "LEDGER"
```

The file ACCTS will now be named LEDGER.

Use NAME to move a file from one directory to another:

```
NAME "\X\CLIENTS" AS "\XYZ\P\CLIENTS"
```

## NEW COMMAND

Deletes the program currently in memory and clears all variables.

### Syntax

NEW

NEW is entered in direct mode to clear memory before entering a new program. GW-BASIC returns to command level after a NEW Command.

NEW closes all files and turns tracing off.

### Example

```
NEW
```

## OCT\$ Function

Returns a string representing the octal value of the decimal argument. X is rounded to an integer before OCT\$(X) is evaluated.

### Syntax

OCT\$(X)

### Example

```
PRINT OCT$(24)
```

returns

```
30
```

See the HEXS function for details on hexadecimal conversion.

## ON COM Statement

Specifies the first line number of a subroutine to be performed when activity occurs on a communications port.

### Syntax

ON COM(*n*) GOSUB *linenumber*

The *n* argument is the number of the communications port.

The *linenumber* is the number of the first line of a subroutine that is to be performed when activity occurs on the specified communications port.

A *linenumber* of zero disables the communications event trap.

The ON COM statement will only be executed if a COM(*n*) ON statement has been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the ON COM statement is not zero, GW-BASIC checks between statements to see if communications activity has occurred on the specified port. If communications activity has occurred, a GOSUB will be performed to the specified line.

If a COM OFF statement has been executed for the communications port, the GOSUB is not performed and is not remembered.

If a COM STOP statement has been executed for the communications port, the GOSUB is not performed, but will be performed as soon as a COM ON statement is executed.

When an event trap occurs (i.e., the GOSUB is performed), an automatic COM STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a COM ON statement unless an explicit COM OFF was performed inside the subroutine.

The RETURN *linenumber* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when GW-BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

## Compiler/Interpreter Differences

With the compiler, you must use the /V or /W option on the compiler command line if a program contains an ON COM statement. These options allow the compiler to function correctly when event trapping routines are included in a program.

## ON ERROR GOTO Statement

Enables error handling and specifies the first line of the error handling routine.

### Syntax

ON ERROR GOTO <line number>

Once error handling has been enabled all errors detected, including direct mode errors such as syntax errors, cause a jump to the specified error handling routine. If the line number does not exist, an "Undefined line" error results.

Use ON ERROR GOTO 0 for errors having no recovery action. Subsequent errors will print the relevant error message and halt the program.

### Note

If an error occurs during an error handling routine its message is printed and the program halted. Error trapping does not occur within the error handling routine.

### Example

```
10 ON ERROR GOTO 1000
```

## ON...GOSUB And ON...GOTO Statement

Branches to one of several specified line numbers, depending on the value of an expression.

### Syntax

```
ON <expression> GOTO <list of line numbers>  
ON <expression> GOSUB <list of line numbers>
```

The value of the expression determines which line number in the list is used for branching. For example, if the value is three, the third line number in the list is the destination. If the value is a noninteger, the number is rounded.

In the ON...GOSUB statement, each line number must be the first line number of a subroutine.

If the value of the expression is zero or greater than the number of items in the list, control drops to the next BASIC statement. If the value of the expression is negative or greater than 255, there is an "Illegal function call".

### Example

```
100 ON L-1 GOTO 150,300,320,390
```

## ON KEY Statement

Specifies the first line number of a subroutine to be performed when a particular key is used.

### Syntax

```
ON KEY(n) GOSUB <line number>
```

(n) is the number of a function, direction or user-defined key; <line number> is the number of the first line of a subroutine. A line number of zero disables the event trap.

There must be a KEY(n) ON statement before an ON KEY statement for event trapping. Once enabled and if the line number in the ON KEY statement is not zero, GW-BASIC checks between statements to see if the specified function, user-defined or cursor direction key has been used. If so, the program branches to a subroutine specified by the GOSUB statement.

If a KEY(n) OFF statement has been made for the specified key the GOSUB is not performed and not remembered.

If a KEY STOP statement has been made for the specified key the GOSUB is not performed until a KEY(n) ON statement is carried out.

With an event trap an automatic KEY(n) STOP ensures that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a KEY(n) ON statement unless there was an explicit KEY(n) off inside the subroutine.

You can use the RETURN line number statement to return to a specific line number from the trapping subroutine. Use this carefully as any GOSUBs, WHILEs, or FORs active at the time of the trap remain so and errors such as "FOR without NEXT" may result.

Event trapping only takes place during a program.

The following rules apply to trapped keys:

- \* Process the line printer echo toggle key first. Defining this key as a user-defined key trap does not prevent characters from being echoed to the line printer if depressed.
- \* Examine the function and cursor direction keys next. Defining a function key or cursor direction key as a user-defined key trap has no effect as the keys are considered pre-defined.
- \* Finally, look at the user-defined keys.
- \* Any key trapped is not passed on as it is not read by BASIC.

## Warning

This may apply to any key, including CTRL-C or system reset (warm boot)! This is a powerful feature considering that it is now possible to prevent accidental Breaking out of a program, or even rebooting the machine.

Note that when a key is trapped it is not remembered. You cannot afterwards use the INPUT or INKEY\$ statements to find which key caused the trap. To assign different functions to particular keys set up a different subroutine for each key.

The ON KEY(n) statement allows 6 additional user-defined KEY traps. You can trap any key — control-key, shift-key, or super-shift-key as follows:

ON KEY(i) GOSUB <line number>

<i> is an integer expressing a legal user-defined key number.

## Example

```
10 KEY 4,"SCREEN 0,0" 'assigns softkey 4
20 KEY(4) ON 'enables event trapping
.
.
70 ON KEY(4) GOSUB 200
.
.
.
```

key 4 used

```
.
.
.
200 'subroutine for the screen.
```

Normal function associated with function key 4, has been replaced with "SCREEN 0,0". This is printed whenever that key is used. The value may be reassigned and resume its standard function when the machine is rebooted.

```
100 KEY 15, CHR$(&H04) + CHR$(31)
105 REM ** Key 15 now is Control-S **
110 KEY(15) ON
```

```
1000 PRINT "If you want to stop processing for a break"
1010 PRINT "press the Control key and the 'S' at the"
1020 PRINT "same time."
1030 ON KEY(15) GOSUB 3000.
```

(where &H04 indicates CONTROL mode set and 31 is the scan code for the S key).

Use CTRL-S.

```
3000 REM ** Suspend processing loop.
3010 CLOSE #1
3020 RESET
3030 CLS
3035 PRINT "Enter CONT to continue."
3040 STOP
3050 OPEN "A", #1, "ACCOUNTS.DAT"
3060 RETURN
```

CTRL-S has been used to enter a subroutine which closes the files and halts the program until ready to continue.

## ON PLAY Statement

Branches to a specified subroutine when the music queue contains fewer than (n) notes. This allows continuous music during a program.

### Syntax

ON PLAY (n) GOSUB LINE-NUMBER

(n) is an Integer expression in the range one to 32. Values outside this range result in an "Illegal function call".  
LINE-NUMBER is the statement line number of the Play event trap subroutine.

ON PLAY causes an event trap when the Background Music queue goes from (n) notes to (n-1) notes.

(n) must be an integer between one and 255

PLAY ON Enables Play event trapping

PLAY OFF Disables Play event trapping

PLAY STOP Suspends Play event trapping

With a PLAY OFF statement the GOSUB is not performed and not remembered.

With a PLAY STOP statement the GOSUB is not performed until a PLAY ON statement is made.

Play event traps are issued only when playing background music (for example PLAY "MB.."). They are not issued when running in Music Foreground (for example default case, or PLAY "MF..").

When an event trap happens through GOSUB an automatic PLAY STOP ensures that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a PLAY ON statement unless an explicit PLAY OFF was performed inside the subroutine.

A play event trap is not issued if the background music queue has already gone from (n) to (n - 1) notes when a PLAY ON is performed. If (n) is large, the number of event traps diminish program speed.

You can use the RETURN line number form to return to a specific line number from the trapping subroutine. Use this carefully as any other GOSUBs, WHILEs, or FORs active at the time of the trap remain and errors such as "FOR without NEXT" may result.

Also see the PLAY ON, PLAY OFF, PLAY STOP statements.

### Example

```
100 PLAY ON
.
.
540 PLAY "MB L1 XZITHER$"
550 ON PLAY(8) GOSUB 6000
.
.
6000 REM **BACKGROUND MUSIC**
6010 LET COUNT% = COUNT% + 1
.
6999 RETURN
```

Control branches to a subroutine when the background music buffer decreases to seven notes.

## ON STRIG Statement

Specifies the first line number of a subroutine to be performed when the joystick trigger is pressed.

### Syntax

ON STRIG(n) GOSUB <line number>

(n) is the number of the joystick trigger; <line number> is the number of the first line of a subroutine. A line number of zero disables the event trap.

A STRIG ON statement must come before an ON STRIG to enable event trapping. If the line number in the ON STRIG statement is not zero GW-BASIC checks between statements to see if the joystick trigger has been pressed. If it has, a GOSUB is performed to the specified line.

After a STRIG OFF statement the GOSUB is neither performed nor remembered.

With a STRIG STOP statement the GOSUB is not carried out until a STRIG ON statement is performed.

When an event trap occurs an automatic STRIG STOP ensures that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a STRIG ON statement unless an explicit STRIG OFF was made inside the subroutine.

You can use the RETURN line number form to return to a specific line number from the trapping subroutine. Use this return carefully as any GOSUBs, WHILEs, or FORs active at the time of the trap remain. Errors such as "FOR without NEXT" may result.

Event trapping only takes place during a program. It is automatically disabled when an error trap occurs.

### **Compiler/Interpreter Differences**

With the compiler, you must use the /V or /W option on the compiler command line if a program contains an ON STRIG statement. These switches allow the compiler to function correctly when you include event trapping routines in a program.

## **ON TIMER Statement**

Provides an event trap during real time.

### **Syntax**

ON TIMER (n) GOSUB <line-number>

ON TIMER causes an event trap every (n) seconds; (n) must be a numeric expression in the range of one to 86400 (one second to 24 hours). Values outside this range generate an "Illegal function call".

A TIMER ON statement must be made before an ON TIMER to enable event trapping. If the line number in the ON TIMER statement is not zero, GW-BASIC checks between statements to see if the time has been reached. If it has, a GOSUB is performed to the specified line.

After a **TIMER OFF** statement **GOSUB** is neither performed nor remembered.

With a **TIMER STOP** statement the **GOSUB** is not performed until a **TIMER ON** statement is carried out.

When an event trap occurs an automatic **TIMER STOP** ensures that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **TIMER ON** statement unless an explicit **TIMER OFF** was made inside the subroutine.

You can use the **RETURN** line number form to return to a specific line number from the trapping subroutine. Use this carefully as any **GOSUBs**, **WHILEs**, or **FORs** active at the time of the trap remain so. Errors such as "FOR without NEXT" may result.

### Example

Display the time of day on line 1 every minute.

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON

.

10000 LET OLDROW = CSRLIN
10010 LET OLDCOL = POS(0) 'Save current Column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'Restore Row & Col
10040 RETURN
```

See also the **TIMER ON**, **TIMER OFF** and **TIMER STOP** Statements.

## OPEN COM Statement

Opens and initializes a communications channel for input/output.

### Syntax

```
OPEN "COM $n$ : [speed] [, [parity] [, [data] [, [stop]  
[, RS] [, CS[ $n$ ] [, DS[ $n$ ]  
[, CD[ $n$ ] [, BIN] [, ASC] [, LF]]]"  
[FOR mode AS [#]filename [LEN = reclen]
```

COM $n$ : is the name of the device to be opened.

The  $n$  argument is the number of a legal communications device, i.e., COM1: or COM2:.

The *speed* is the baud rate, in bits per second, of the device to be opened.

The *parity* designates the parity of the device to be opened. Valid entries are: N (none), E (even), O (odd), S (space), or M (mark).

The *data* argument designates the number of data bits per byte. Valid entries are: 5, 6, 7, or 8.

The *stop* argument designates the stop bit. Valid entries are: 1, 1.5, or 2.

RS suppresses RTS (Request To Send).

CS( $n$ ) controls CTS (Clear To Send).

DS( $n$ ) controls DSR (Data Set Ready).

CD( $n$ ) controls CD (Carrier Detect).

LF allows communication files to be printed on a serial line printer. When LF is specified, a linefeed character (0AH) is automatically sent after each carriage return character (0CH). This includes the carriage return sent as a result of the width setting. Note that INPUT and LINE INPUT, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed.

The LF option is superseded by the BIN option.

BIN opens the device in binary mode. BIN is selected by default unless ASC is specified.

In the BIN mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and CONTROL-Z is not treated as end-of-file. When the channel is closed, CONTROL-Z will not be sent over the RS232 line. The BIN option supersedes the LF option.

ASC opens the device in ASCII mode. In ASCII mode, tabs are expanded, carriage returns are forced at the end-of-line, CONTROL-Z is treated as end-of-file, and the XON/XOFF protocol is enabled. When the channel is closed, CONTROL-Z is sent over the RS-232 line.

The *mode* argument is one of the following string expressions:

OUTPUT Specifies sequential output mode.

INPUT Specifies sequential input mode.

If the *mode* expression is omitted, it is assumed to be random input/output. Random input/output cannot, however, be explicitly chosen as *mode*.

The *filenumber* is the number of the file to be opened. The OPEN COM statement must be executed before a device can be used for RS-232 communication. Any syntax errors in the OPEN COM statement will result in a "Bad File name" error.

The *speed*, *parity*, *data*, and *stop* options must be listed in the order shown in the above syntax. The remaining options may be listed in any order, but they must be listed after the *speed*, *parity*, *data*, and *stop* options.

A "Device timeout" error occurs if Data Set Ready (DSR) is not detected.

### Example

The following opens communications channel 1 in random mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in the binary mode. Other lines in the program may now access channel 1 as file number 2.

```
10 OPEN "COM1:9600,N,8,1,BIN" AS 2
```

## OPEN Statement

Allows I/O to a file.

### Syntax

```
OPEN <mode1> [, [#] <file number> , <filespec>  
[ , <record length> ]  
OPEN <filespec> (FOR <mode2> ) AS (access)  
[#] <file number> [LEN = <record length> ]
```

The filespec is an optional disk specification followed by a conforming filename or pathname.

<mode1> is a string expression. The first character must be one of the following specifications for;

- O sequential output mode.
- I sequential input mode.
- R random input/output mode.
- A sequential output mode; sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement then extends the file.

In <mode2> OUTPUT, INPUT and APPEND are used in the same way. If omitted the default random access mode is assumed. This cannot be expressed explicitly as the file mode.

A difference between this version of the interpreter and the 2.0 version is the addition of the *access* clause to support networking. The *access* argument can be one of the following:

DEFAULT: If access is not specified, the file may be opened for reading and writing any number of times by this process, but other processes are denied access to the file while it is opened.

SHARED: Any process on any machine may read from or write to this file.

LOCK READ: No other process is granted read access to this file.

This access is granted only if no other process has a previous LOCK READ access to the file.

**LOCK WRITE:** No other process is granted write access to this file.

This access is granted only if no other process has a previous access of this kind to the file.

**LOCK READ WRITE:** No other process is granted either read or write access to this file. Again, this access is only granted if LOCK READ WRITE has not already been granted to another process.

<file number> is an integer expression whose value is between 1 and 255. The number remains with the OPENed file and is used to refer other disk I/O statements to the file.

<record length> is an integer expression setting the record length for random files. GW-BASIC ignores this option if used in a statement to OPEN a sequential file. The default length for records is 128 bytes, unless the command line options /I and /R have been used (See Section 1.2 Command Line Option Switches).

A file must be opened before any I/O operation. OPEN allocates a buffer for I/O to the file and determines the mode of access used with the buffer.

With OPEN pathname can be used instead of filespec. If used with a drive this must be specified at the beginning of the pathname.

### Example

```
B:\SALES\JOHN"
```

```
"\SALES\B:JOHN"
```

The first statement is legal; the second isn't.

The LEN = option is ignored if the file being opened has been specified as a sequential file.

It is possible to reference the same file in a sub-directory via different paths and nearly impossible for BASIC to know that it is the same file simply by looking at the path. You can not therefore open the file for OUTPUT or APPEND on the same disk even if the path is different.

## Example

if MARY is your current directory, then:

```
OPEN "REPORT" ...  
OPEN "\\SALES\MARY\REPORT" ...  
OPEN "..\MARY\REPORT" ...  
OPEN "...\\MARY\REPORT" ...
```

all refer to the same file.

Note that a file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output on only one file number at a time.

## Examples

```
10 OPEN "I",2,"INVEN"  
10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

If a device called FOO is written and installed then the OPEN statement might appear as:

```
10 OPEN "\\DEV\FOO" FOR OUTPUT AS #1
```

To open the printer for output, use the line:

```
100 OPEN "LPT:" FOR OUTPUT AS #1
```

which uses the GW-BASIC device driver, or as part of a pathname as in:

```
100 OPEN "\\DEV\LPT1" FOR OUTPUT AS #1
```

which uses the MS-DOS device driver.

## OPTION BASE Statement

Declares the minimum value for array subscripts.

### Syntax

OPTION BASE(n)

n is one or zero. The default base is zero. If the statement OPTION BASE 1 is carried out the lowest value an array subscript can have is one.

Code the OPTION BASE statement before defining or using any arrays.

Chained programs can have an OPTION BASE statement if no arrays are passed between them or the specified base is identical in the chained programs. The chained program inherits the OPTION BASE value of the chaining program.

### Example

```
10 OPTION BASE 1
```

## OUT Statement

Sends a byte to a machine output port.

### Syntax

OUT I,J

I is the port number and an integer expression in the range zero to 65535. J is the data to be transmitted. It must be an integer expression in the range zero to 255.

### Example

```
100 OUT 12345,255
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,12345  
MOV AL,255  
OUT DX,AL
```

## PAINT Statement

Fills a graphics area with the colour or pattern specified.

### Syntax

PAINT (<x>,<y>)[,<paint attribute> [,<border colour>] [,<background attribute>]]

(<xstart>) and <ystart> are the starting coordinates. Painting should always start on a non-border point. If painting starts within a border, the bordered figure is painted; if started outside a bordered figure, the background is painted.

If the paint attribute is a string expression PAINT will "Tile", a process similar to "Line-styling". PAINT looks at a "tiling" mask each time a point is put down on the screen. When paint attribute is a numeric expression, the number must be a valid colour; if not specified, the foreground colour is used.

<border colour> identifies the figure to be filled. When the border colour is encountered, painting of the current line stops. If the border colour is not specified, the paint attribute is used.

<background attribute> is a string formula returning character data. When omitted, the default is CHR\$(0).

When specified, the background attribute gives the "background tile slice" to skip when checking for boundaries. Painting is finished when adjacent points display the paint colour. Specifying a background tile slice allows over-painting without ending the process because two consecutive lines with the same paint attributes are encountered.

Painting is complete when the entire line is equal to the paint colour.

The PAINT command can fill any figure, but painting complex figures may result in an "Out of Memory" error. Use the CLEAR statement to increase the amount of stack space available.

The PAINT command allows coordinates outside the screen or viewport.

## Tiling

Tiling is a PAINT pattern design 8 bits wide and up to 64 bytes long. Each byte in the Tile String masks 8 bits along the x axis when putting down points.

A Tile mask works as follows:

Use the syntax PAINT (x,y), CHR\$(n)...CHR\$(n) where (n) is a number between zero and 255 represented in binary across the x-axis of the "tile". Each CHR\$(n) up to 64 generates an image of the bit arrangement of the code for the assigned character. For example, the decimal number 85 is binary "01010101"; the graphic image line on a black and white screen generated by CHR\$(85) is an eight pixel line, with even numbered points turned white, and odd ones black. Each bit containing a "1" sets the associated pixel on and each bit filled with a "0" sets the associated bit off in a black and white system. The ASCII character CHR\$(85), which is "U", is not displayed in this case. If the current screen mode supports only two colours, then the screen can be painted with 'X's with the following statement.

```
PAINT (320,100),CHR$(129)+CHR$(66)
+CHR$(36)+CHR$(24)+CHR$(24)
+CHR$(36)+CHR$(66)+CHR$(129)
```

This appears on the screen as:

x increases →							
0,0	x					x	CHR\$(129) Tile byte 1
0,1		x				x	CHR\$(66) Tile byte 2
0,2			x			x	CHR\$(36) Tile byte 3
0,3				x	x		CHR\$(24) Tile byte 4
0,4				x	x		CHR\$(24) Tile byte 5
0,5			x			x	CHR\$(36) Tile byte 6
0,6		x				x	CHR\$(66) Tile byte 7
0,7	x					x	CHR\$(129) Tile byte 8

Specifying more than two consecutive bytes (in the tile background slice) that match the tile string, results in an "Illegal function call".

## Example

```
10 PAINT (5,15),2,0
```

begins painting at coordinates 5,15 with colour 2 and border colour 0, and fills a border.

## PALETTE, PALETTE USING Statements

Changes the colours in the palette.

### Syntax

PALETTE [ <colour number> , <display colour> ]

or

PALETTE USING <array> ( <array index> )

<colour number> is a colour number in a GW-BASIC statement, used by the COLOR statement; <display colour> number is an integer representing a colour that can be created on the display screen. The number of colours available (up to 16) depends on which Apricot computer you have.

For example, you may not be able to display all 16 colours at a time but you can choose which colours you want in your palette. See your Apricot technical reference manual for full details.

<array> is the the name of an integer; <array index> specifies the index of the first array element to use.

The palette is a map specifying how colour numbers used in GW-BASIC statements are mapped to colours on the display screen.

The colour number can also be used in the COLOR statement to set the default text colour.

Display colour numbers show how coloured graphic points appear on the display screen. Under a common initial palette setting points coloured with colour number zero appear as black on the display screen. Use the PALETTE statement to change the mapping of colour number zero to display colour white.

If the parameters are not specified, the original palette setting is used.

With the USING option each entry in the palette can be modified at once. GW-BASIC sets each colour number in the palette to the respective display colour in the array.

If the display colour number or an array entry is - 1 the mapping for the associated colour number is not changed. Other negative numbers are illegal values.

If the options are specified, the palette number colour is changed to the colour number. As an example, the current palette consists of display colours 0, 1, 2, and 3. The statement `PALETTE 3,2` results in a new palette of display colours 0, 1, 2, and 2.

### Example

```
PALETTE 0,2
```

changes all points coloured with colour number 0 to display colour 2.

```
PALETTE 0,—1
```

does not modify the palette.

```
PALETTE A%(0)
```

changes each palette entry. All colour numbers are mapped to display colour zero as the array is initialised to zero when first declared. The screen appears as one single colour. Statements can still be entered.

### Example

```
PALETTE
```

sets each palette entry to the appropriate initial display colour. The screen no longer appears as one single colour.

## PCOPY Statement

Copies one screen page to another in all screen modes.

### Syntax

`PCOPY sourcepage, destinationpage`

The *sourcepage* is an integer expression in the range 0 to *n*, where *n* is determined by the current video memory size and the size per page for the current screen mode.

The *destinationpage* has the same requirements as the *sourcepage*. See also `CLEAR`, `SCREEN`.

### Example

This copies the contents of page 1 to page 2:

```
PCOPY 1, 2
```

## PEEK Function

Returns the byte read from the indicated memory location (I).

### Syntax

PEEK(I)

The returned value is an integer in the range zero to 255. I must be in the range -32768 to 65535. I is the offset from the current segment defined by the last DEF SEG statement. For the interpretation of a negative value of I, see VARPTR function.

PEEK complements the POKE statement.

### Example

```
A = PEEK(&H5A00)
```

The value at the location with the hex offset 5A00 is loaded into a variable, A.

## PLAY Statement

Plays music as specified by the <string> subcommand.

### Syntax

PLAY <string>

PLAY is similar to DRAW in use. It embeds a Music Macro Language into one statement.

The following subcommands, used as part of the PLAY statement, specify the particular action .

#### ***Prefixes-ChangeOctave***

- > Increments octave. Octave does not go beyond six.
- < Decrements octave. Octave does not drop below zero.

## ***Tone***

O <n> Sets the current octave. There are seven octaves, numbered zero through six.

A-G Plays a note in the range A-G. # or + after the note specifies sharp; - specifies flat.

N <n> Plays note n. n ranges from zero through 84 (in the seven possible octaves there are 84 notes). n = zero means a rest.

## ***Duration***

L <n> Sets the length of each note. L 4 is a quarter note, L 1 is a whole note, and so on. n may be in the range one through 64. The length may follow the note when only a change of length is wanted for a particular note. In this case, A 16 is equivalent to L 16 A.

MN Sets "music normal" so that each note plays 7/8 of the time determined by the length (L).

ML Sets "music legato" so that each note plays the full period set by length (L).

MS Sets "music staccato" so that each note plays 3/4 of the time determined by the length (L).

## ***Tempo***

P <n> Specifies a pause, ranging from one through 64. This option corresponds to the length of each note, set with L <n>.

T <n> Sets the "tempo," or the number of L 4's in one minute. n ranges from 32 through 255. The default is 120.

## ***Operation***

MF Sets music (PLAY statement) and SOUND to run in the foreground. Each subsequent note or sound does not start until the previous note or sound has finished. This is the default setting.

MB Music (PLAY statement) and SOUND are set to run in the background. Each note or sound is placed in a buffer and lets the program continue while the note or sound plays in the background. The number of notes that can be played at one time depends on the particular machine.

## Substring

X <string> Performs a substring. Because of the slow clock interrupt rate, some notes will not play at higher tempos, L 64 at T 255, for example.

Note that a substring may be carried out through adding the character form of the address to "X".

## Suffixes

# or + Follows a specified note, and turns it into a sharp or flat.

A full stop after a note makes the note play  $3/2$  times the length determined by L multiplied by T (tempo). Multiple full stops may appear after a note. The full stop is scaled accordingly; for example, A. is  $3/2$ , A.. is  $9/4$ , A... is  $27/8$ , and so on.

Full stops may appear after a pause (P). The pause length can then be scaled in the same way as notes.

## Examples

```
PLAY "<<" 'Decrement by two octaves
```

```
PLAY ">" 'Increment by an octave
```

```
PLAY ">A" 'Increment by an octave and play  
an A note.
```

```
PLAY "XSONG$"
```

```
LET LISTEN$ = "T180 O2 P2 P8 L8 GGG  
L2 E-"  
LET FATE$ = "P24 P8 L8 FFF L2 D"  
PLAY LISTEN$ + FATE$
```

This plays the beginning of the first movement of Beethoven's Fifth Symphony.

## PLAY Function

Returns the current number of notes in the background music queue.

### Syntax

PLAY (n)

(n) is a dummy argument and may be any value.

PLAY(n) returns zero when in Music Foreground Mode.

## PLAY ON, PLAY OFF, PLAY STOP Statements

PLAY ON enables play event trapping.

PLAY OFF disables play event trapping.

PLAY STOP suspends play event trapping.

### Syntax

PLAY ON

PLAY OFF

PLAY STOP

After a PLAY OFF statement the GOSUB is not performed and not remembered.

After a PLAY STOP statement the GOSUB is not carried out until after a PLAY ON .

When an event is trapped an automatic PLAY STOP ensures that recursive traps cannot take place.

The RETURN from the trapping subroutine performs a PLAY ON statement unless an explicit PLAY OFF was performed inside the subroutine.

Use the RETURN line number form to return to a specific line number from the trapping subroutine. Use this carefully as other GOSUBs,WHILEs or FORs active at the time of the trap remain so. Errors such as "FOR without NEXT" may result.

## PMAP Function

Maps world coordinate expressions to physical locations and vice versa.

### Syntax

PMAP <expression> , <function>

There are four map functions as follows :

- \* 0 Maps world expression to physical x coordinate.
- \* 1 Maps world expression to physical y coordinate.
- \* 2 Maps physical expression to world x coordinate.
- \* 3 Maps physical expression to world y coordinate.

Use these to find equivalent point locations between the world coordinates created with the WINDOW statement and the physical coordinate system of the screen or viewport as defined by the VIEW statement.

### Examples

With a WINDOW SCREEN (80,100) - (200,200) the upper left coordinate of the window is (80,100) and the lower right is (200,200). The screen coordinates can be (0,0) in the upper left hand corner and (639,199) in the lower right.

Then;

```
X = PMAP(80,0)
```

returns the screen x coordinate of window x coordinate 80:

```
0
```

The PMAP function in the statement:

```
Y = PMAP(200,1)
```

returns the screen y coordinate of the window y coordinate 200

```
199
```

The PMAP function in the statement:

```
X = PMAP(619,2)
```

returns the "world" coordinate corresponding to the screen or viewport x coordinate 619:

```
199
```

The PMAP function in the statement:

```
Y = PMAP(100,3)
```

returns the "world" y coordinate corresponding to the screen or viewport y coordinate 100:

```
140
```

## POINT Function

Reads the colour number of a pixel from the screen, also retrieves the current graphics cursor coordinates

### Syntax

POINT (<xcoordinate> , <ycoordinate> of the pixel to be referenced.

or

POINT (<function> )

POINT with one argument allows the retrieval of the current graphics cursor coordinates. For example, x = POINT(func), returns the value of the current x or y Graphics accumulator as follows:

function =

- \* 0 Returns the current physical x coordinate.
- \* 1 Returns the current physical y coordinate.
- \* 2 Returns the current logical x coordinate. If the WINDOW statement has not been used the same value as the POINT(0) function is returned.
- \* 3 Returns the current logical y coordinate if WINDOW is active. Otherwise returns the current physical y coordinate.

The physical coordinate is on the screen or current viewport.

### Examples

```
10 SCREEN 1
20 LET C = 3
30 PSET (10,10),C
40 IF POINT(10,10) = C THEN PRINT "This
point is colour ";C
```

```
5 SCREEN 2
10 IF POINT(i,i) < > 0 THEN PRESET (i,i) ELSE PSET
(i,i) 'invert current state of a point.
20 PSET (i,i),1 - POINT(i,i) 'another way to invert a
point if the system is black and white.
```

## POKE Statement

Writes a byte into a memory location.

### Syntax

POKE I,J

I and J are integer expressions. I represents the offset of the memory location and J is the data byte. J must be in the range zero to 255.

I must be in the range - 32768 to 65535. I is the offset from the current segment set by the last DEF SEG statement.

See VARPTR Function for interpretation of I's negative values.

PEEK complements POKE .

### Warning

Use POKE carefully. It can cause GW-BASIC or MS-DOS to crash.

### Example

```
10 POKE &H5A00,&HFF
```

## POS Function

Returns the current horizontal (column) position of the cursor.

### Syntax

POS(I)

The leftmost position is one. I is a dummy argument. Use CSRLIN to return the current vertical line position of the cursor.

### Example

```
IF POS(X) > 60 THEN BEEP
```

Also see LPOS Function.

## PRESET Statement

Draws a specified point on the screen.

### Syntax

PRESET [STEP] (<xcoordinate> , <ycoordinate> )  
[, <colour> ]

<xcoordinate> and <ycoordinate> specify the the pixel to be set; <colour> is the colour number to be used for the specified point.

Coordinates can be shown as absolutes or the STEP option can be used to reference a point relative to the most recent point used. For example, if the most recent point referenced was (10,10), STEP (10,5) would reference a point at (20,15).

PRESET works like PSET except that the background colour is selected if no colour is specified.

No action is taken if a coordinate is outside the current viewport. Nor is a error message given.

## Example

```
5 REM DRAW A LINE FROM (0,0) TO  
  (100,100)  
10 FOR i=0 TO 100  
20 PRESET (i,i),1  
30 NEXT  
  
35 REM NOW ERASE THAT LINE  
40 FOR i=0 TO 100  
50 PRESET STEP (-1,-1)  
60 NEXT
```

After a line is drawn from (0,0) to (100,100) it is erased by overwriting with the background colour.

## PRINT Statement

Outputs data on the screen.

### Syntax

PRINT [ <list of expressions> ]

If the list of expressions is omitted, a blank line is printed. If included, the values of the expressions are printed on the screen. The expressions can be numeric or string. String literals must be enclosed in quotation marks.

Use a question mark (?) as a short form. It is treated as the word PRINT, and appears as such in subsequent listings.

### *Print Positions*

Punctuation separates the items in the list. The line is divided into print zones of 14 spaces each. After a comma the value is printed at the start of the next zone; with a semi-colon the value is printed after the previous one.

Spaces between expressions give the same result as a semicolon.

When a comma or a semicolon ends the list of expressions, the next PRINT statement begins on the same line. If there isn't a comma or semicolon, a carriage return is printed at the end of the line. If the printed line is wider than the screen width, printing continues on the next physical line.

A space follows printed numbers.  
A space comes before a positive number.  
A minus comes before negative numbers.

Single precision numbers that can be represented accurately with 6 or fewer digits in the unscaled format are output using this format. For example, 1 E-7 is output as .00000 and 1 E-8 is output as 1 E-08.

Double numbers that can be represented accurately with 16 or fewer digits in the unscaled format are output using this format.

For example, 1 D-15 is output as .00000000000000001 and 1 D-16 is output as 1 D-16.

### Example

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
```

returns

```
10    0    -25    3125
```

The commas in the PRINT statement mean that each value is printed at the beginning of the next print zone.

### Example

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
```

returns

```
? 9
9 SQUARED IS 81 AND 9 CUBED IS 729
? 21
21 SQUARED IS 441 AND 21 CUBED IS 9261
?
```

The semicolon at the end of line 20 means that both PRINT statements are printed on the same line. With Line 40 a blank line is printed before the next prompt.

### Example

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
60 PRINT
```

returns

```
5 10 10 20 15 30 20 40 25 50
```

The semicolons mean that each value is printed immediately after the previous one. In Line 40 a question mark is used instead of PRINT.

**Note:** Escape sequences will not be interpreted by GW-BASIC. However, programmers who wish to use escape sequences may do the following:

Instead of entering the MS-BASIC statement:-  
Print CHR\$(27),+"E",  
at the beginning of the program enter,

```
OPEN "O",#1,"CONS:"
```

To print text and escape sequences to the screen, enter,

```
PRINT #1, CHR$(27),+"E".
```

It is, however, recommended that the programmer make use of the "LOCATE" and "COLOR" statements rather than depending on escape sequences.

## PRINT USING Statement

Prints strings or numbers using a specified format.

### Syntax

**PRINT USING** <string exp>; <list of expressions>

<list of expressions> are the string or numeric expressions to be printed, separated by semicolons.

<string exp> is a string literal or variable of special formatting characters. These determine the field and format of the printed strings or numbers.

### String Fields

When **PRINT USING** is used for strings, a formatting character is used:

**"!"** Specifies that only the first character in the given string is to be printed.

**"\n spaces\** Specifies that 2 + n characters from the string are to be printed. If there are no spaces with the backslash, two characters are printed; with one space three characters are printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

### Example

```
10 A$="LOOK":B$="OUT"  
30 PRINT USING "!";A$;B$  
40 PRINT USING "\n";A$;B$  
50 PRINT USING "\n";A$;B$;"!!"
```

returns

```
LO  
LOOKOUT  
LOOK OUT !!
```

**"&"** Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

### Example

```
10 A$="LOOK":B$="OUT"  
20 PRINT USING "!";A$;  
30 PRINT USING "&";B$
```

returns

```
LOUT
```

### *Numeric Fields*

When PRINT USING is used for numbers, the following special characters can format the numeric field:

# A number sign represents each digit position. These positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

A decimal point can be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit is printed (as 0, if necessary). Numbers are rounded as necessary.

### Example

```
PRINT USING "###.##";.78  
0.78
```

```
PRINT USING "###.##";987.654  
987.65
```

```
PRINT USING "###.##";10.2,5.3,66.789,234  
10.20 5.30 66.79 0.23
```

In this last example three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string means the sign of the number (plus or minus) is printed before or after the number.

– A minus sign at the end of the format field means negative numbers are printed with a trailing minus sign.

### Example

```
PRINT USING "+###.##";-68.95,2.4,55.6,-.9  
-68.95 + 2.40 + 55.60 -0.90
```

```
PRINT USING "###.##- ";-68.95,22.449,- 7.01  
68.95 - 22.45 7.01 -
```

\*\* This at the beginning of the format string means leading spaces in the numeric field are filled with asterisks. The \*\* also specifies positions for two more digits.

### Example

```
PRINT USING "**].#";12.39,-0.9,765.1  
*12.4 *-0.9 765.1
```

\$\$ This means a dollar sign is printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$\$. Negative numbers cannot be used unless the minus sign trails to the right.

### Example

```
PRINT USING "$$###.##";456.78  
$456.78
```

\*\*\$ This at the beginning of a format string combines the effects of the above two symbols. Leading spaces are asterisk-filled and a dollar sign printed before the number. \*\*\$ specifies three more digit positions, one of which is the dollar sign.

The exponential format cannot be used with \*\*\$. When negative numbers are printed, the minus sign appears immediately to the left of the dollar sign.

### Example

```
PRINT USING "***$##.##";2.34  
***$2.34
```

, A comma to the left of the decimal point in a formatting string means a comma is printed to the left of every third digit on the left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. It has no effect if used with exponential (^^^^) format.

### Example

```
PRINT USING "####.##";1234.5  
1,234.50
```

```
PRINT USING "####.##,";1234.5  
1234.50,
```

^^^^ Four carets (or up-arrows) can be placed after the digit position characters to specify exponential format. They allow space for E + xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.

### Example

```
PRINT USING "##.##^^^^";234.56  
2.35E + 02
```

```
PRINT USING ".####^^^^-";-888888  
-.8889E + 06
```

```
PRINT USING "+ ##^";123  
+.12E+03
```

— This sign in the format string means the next character is output as a literal character.

### Example

```
PRINT USING " _ !##.##!";12.34  
!12.34!
```

The literal character can be an underscore by placing " \_ " in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding means the number exceeds the field, a percent sign is printed in front of the rounded number.

### Example

```
PRINT USING "##.##";111.22  
%111.22
```

```
PRINT USING ".##";.999  
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error results.

## PRINT# and PRINT# USING Statements

Writes data to a sequential file.

### Syntax

`PRINT#<file number>,[USING <string exp>;] <list of expressions>`

<file number> is the number used when the file was opened for output. <string exp> contains the formatting characters described in PRINT USING. The expressions in the list are the numeric and string expressions to be written to the file.

PRINT# does not compress data. An image of the data is written to the file as it would be displayed on the terminal screen with a PRINT statement. For this reason, delimit the data carefully, so it is input correctly.

(See WRITE# statement.)

In the list of expressions, numeric expressions are usually delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

If commas are used as delimiters, the extra blanks inserted between print fields are also written to the file.

String expressions must be separated by semicolons. To format the string expressions correctly in the file use explicit delimiters in the list of expressions.

## Example

If:

A\$="CAMERA" and B\$="93604-1".

The statement PRINT#1,A\$;B\$ writes CAMERA93604-1 to the file. As there are no delimiters, this cannot be read back as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;" ";B$
```

The image written to the file is CAMERA,93604-1 which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or linefeeds, write them to the file surrounded by explicit quotation marks, CHR\$(34).

## Example

If:

A\$="CAMERA, AUTOMATIC" and B\$="93604-1".

The statement PRINT#1,A\$;B\$ writes the following image to file:

```
CAMERA, AUTOMATIC 93604-1
```

The statement INPUT#1,A\$,B\$ inputs "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly in the file, write double quotation marks to the file image using CHR\$(34).

The statement PRINT#1,CHR\$(34);A\$; CHR\$(34); CHR\$(34);B\$;CHR\$(34) writes the following image to the file:

```
"CAMERA, AUTOMATIC" "93604-1"
```

The statement INPUT#1,A\$,B\$ inputs "CAMERA, AUTOMATIC" to A\$ and "93604-1" to B\$.

The PRINT# statement can also be used with the USING option to control the format of the file.

## Example

```
PRINT#1,USING"$$###.##,";J;K;L
```

## PSET Statement

Draws a specified point on a screen.

### Syntax

PSET [STEP] ( <xcoordinate> , <ycoordinate> )  
[ , <colour> ] STEP ( <offset> , <offset> )

( <xcoordinate> and <ycoordinate> ) specify the point on the screen to be coloured; <colour> is the number of the colour to be used.

With the STEP option the given x and y coordinates are relative, not absolute. This means x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate. For example, if the most recent point referenced was (0,0), PSET STEP (10,0) would reference a point at offset 10 from x and offset 0 from y.

When GW-BASIC scans coordinate values, it allows them beyond the edge of the screen. The size of the screen depends on the machine being used and can be adjusted with the WIDTH statement. However, values outside the integer range - 32768 to 32767 cause an "Overflow" error.

The coordinate (0,0) is always the upper left corner of the screen.

With PSET the colour can be omitted from the command line. The default is then the foreground colour.

### Example

```
5 REM DRAW A LINE FROM (0,0) TO  
  (100,100)  
10 FOR I=0 TO 100  
20 PSET (I,I)  
30 NEXT I  
  
35 REM NOW ERASE THAT LINE  
40 FOR I=0 TO 100  
50 PSET STEP (-1,-1),0  
60 NEXT I
```

Here a line is drawn from (0,0) to (100,100) and then erased through overwriting with the background colour.

## PUT Statement - File I/O

Writes a record from a random buffer to a random access file.

### Syntax

PUT [#] <file number> [, <record number> ]

<file number> is the number of the opened file.

<record number> is the position which the record will occupy. If the <record number> is omitted, the next available record number (after the last PUT or GET) is assumed. The largest possible record number is 16,777,215 and the smallest one.

### Note

LSET, RSET, PRINT#, PRINT# USING, and WRITE# can be used to put characters in the random file buffer before a PUT statement.

With WRITE#, GW-BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer leads to a "Field overflow" error.

For details on file I/O, see Section 2, Working with Files and Devices.

### Example

```
100 PUT 1, A$, B$, C$
```

## PUT Statement - Graphics

The GET and PUT statements are used together to transfer graphic images to and from the screen. The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The PUT statement transfers the image stored in the array onto the screen.

### Syntax

PUT (x1,y1), <array name> [,action verb]

used with

GET (x1,y1)-(x2,y2), <array name>

(x1,y1) in the PUT statement specifies the point where a stored image is to be displayed on the screen. The specified point is the coordinate of the top left corner of the image. If the projected image is too large for the current viewport, an "Illegal function call" results.

The action verbs are PSET, PRESET, AND, OR, XOR.

PSET transfers the data point by point onto the screen. Each point has the exact colour attribute as when taken from the screen with a GET.

PRESET is the same as PSET except it produces a complementary image.

AND transfers the image over an existing one. Points having the same colour in both the existing image and the PUT image remain the same colour; points not having the same colour in both the existing image and the PUT image do not.

OR superimpose the image onto an existing image.

XOR is a special mode often used for animation. The points on the screen can be inverted to where a point exists in the array image. This works like the cursor. When an image is PUT against a complex background twice, the background is restored unchanged. An object can then be moved around the the screen without the background being erased.

XOR is the default action verb.

GET and PUT are very useful for animation. This works as follows:

- 1\* PUT the object on the screen.
- 2\* Recalculate the new position of the object.
- 3\* PUT the object on the screen a second time at the old location (using XOR) to remove the old image.
- 4\* Go to step 1, but this time PUT the object at the new location.

Cut down flicker by minimising the time between steps 4 and 1 and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, they should be processed at once, one step at a time.

Use PSET if it is not important to preserve the background. This leaves a border around the image as large or larger than the maximum distance the object will move. When it is moved, the border erases any points left by the previous PUT. This method is faster than the XOR method since only one PUT is necessary to move an object. However you must PUT a larger image.

You can examine the x and y dimensions and even the data if an integer array is used. With the interpreter, the x dimension is in element zero of the array, and the y is found in element one. Remember that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

## RANDOMIZE Statement

Reseeds the random number generator.

### Syntax

RANDOMIZE [ <expression> ]

If <expression> is omitted, GW-BASIC suspends the program and asks for a value by printing Random Number Seed ( - 32768 to 32767)? before carrying out RANDOMIZE.

If the expression is a variable, its value is used to seed the random numbers. If the expression is TIMER then this function is used to pass a random number seed.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

### Example

```
10 RANDOMIZE  
20 FOR I=1 TO 5  
30 PRINT RND;  
40 NEXT I
```

returns

Random Number Seed ( - 32768 to 32767)? 3

(You type 3) returns

.885982 .4845668 .586328 .1194246 .7039225

Random Number Seed ( - 32768 to 32767)? 4

(You type 4 for new sequence) returns

.803506 .1625462 .929364 .2924443 .322921

Random Number Seed ( - 32768 to 32767)? 3

This same sequence as the first run returns

.885982 .4845668 .586328 .1194246 .7039225

Note that the numbers your program produces may not be the same as the ones shown.

## READ Statement

Reads values from a DATA statement and assigns them to variables. (See DATA Statement )

### Syntax

READ <list of variables>

A READ statement must always be used with one or more DATA statements. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they don't a Syntax error results.

A single READ statement can access one or more DATA statements, in order, or several READ statements can access the same DATA statement. If the number of variables exceeds the number of elements in the DATA statements, an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statements subsequent READ statements begin reading data at the first unread element. If there are no further READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement .

### Example

```
.  
. .  
80 FOR I=1 TO 10  
90 READ A(I)  
100 NEXT I  
110 DATA 3.08,5.19,3.12,3.98,4.24  
120 DATA 5.08,5.55,4.00,3.16,3.37  
. .  
.
```

This program segment READs the values from the DATA statements into the array A.  
Afterwards, the value of  $A(1) = 3.08$ ,  $A(2) = 5.19$  and so on.

## Example

```
10 PRINT "TOWN", "COUNTY", "POSTCODE"  
20 READ T$,C$,P$  
30 DATA "ABERDEEN", "GRAMPIAN", "AB1 2RN"  
40 PRINT T$,C$,P$
```

returns

```
TOWN COUNTY POSTCODE  
ABERDEEN GRAMPIAN AB1 2RN
```

This program reads string and numeric data from the DATA statement in line 30.

## REM Statement

Allows explanatory remarks to be inserted in a program.

### Syntax

REM <remark>

REM statements are not carried out but are output as entered when the program is listed. They may be branched into from a GOTO or GOSUB statement and the program continues with the first executable statement after the REM statement.

Add remarks to the end of a line by preceding them with a single quotation mark instead of REM.

### Important

Do not use the single quotation form of REM in a data statement as it is considered legal data.

## Example

```
.  
.   
.   
120 REM CALCULATE AVERAGE VELOCITY  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)  
.   
.   
. 
```

or

```
.   
.   
120 FOR I=1 TO 20 'CALCULATE AVERAGE  
VELOCITY  
130 SUM=SUM + V(I)  
140 NEXT I  
.   
.   
. 
```

## RENUM Command

Renumbers program lines.

### Syntax

RENUM [[<new number>][,<old number>][,  
<increment>]]

<new number> is the first line number used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering begins. The default is the first line of the program. <increment> is the line number increment used in the new sequence. The default is 10.

RENUM changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number in xxxxx" is printed. The incorrect line number reference is not changed by RENUM.

### Note

You cannot use RENUM to change the order of program lines or to create line numbers greater than 65529. An "Illegal function call" results.

### Example

```
Renum
```

Renumbers the entire program. The first new line number will be 10. Lines are numbered in increments of 10.

```
RENUM 300,,50
```

Renumbers the entire program. The first new line number will be 300. Lines are numbered in increments of 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20.

## RESET Command

Closes all files.

### Syntax

RESET

With RESET all open files are closed and blocks in memory are written to the disk. This ensures that all files are properly updated in case of machine power loss.

All files must be closed before a disk is removed from its drive.

### Example

```
998 RESET
999 END
```

## RESTORE Statement

Allows DATA statements to be reread from a specified line.

### Syntax

RESTORE [ <line number> ]

After a RESTORE statement without a specified line number the next READ statement accesses the first item in the first DATA statement in the program.

If a line number is specified, the next READ statement accesses the first item in the specified DATA statement.

### Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
```

## RESUME Statement

Continues the program after an error recovery procedure.

### Syntax

```
RESUME  
RESUME 0  
RESUME NEXT  
RESUME <line number>
```

Use any of the syntaxes, depending where the program is to resume.

RESUME	Resumes at the statement that caused the error.
or	
RESUME 0	
RESUME NEXT	Resumes at the statement after the one that caused the error.
RESUME	Resumes at that line number.
< line number >	

A RESUME statement not in an error handling routine leads to a "RESUME without error" message.

### Example

```
10 ON ERROR GOTO 900  
.  
.  
.  
900 IF (ERR = 230) AND (ERL = 90) THEN PRINT  
"TRY AGAIN":RESUME 80  
.  
.  
.
```

## RETURN Statement

See GOSUB...RETURN Statements.

## RIGHT\$ Function

Returns the rightmost I characters of string X\$.

### Syntax

RIGHT\$(X\$,I)

If I is equal to the number of characters in X\$ (LEN(X\$)), returns X\$. If I = 0, the null string (length zero) is returned.

### Example

```
10 A$ = "DISK BASIC"  
20 PRINT RIGHT$(A$,5)
```

returns

```
BASIC
```

See also the LEFT\$ and MID\$ functions.

## RMDIR Statement

Removes an existing directory.

### Syntax

RMDIR <pathname>

<pathname> is the name of the directory to be deleted and it must be a string of less than 128 characters. RMDIR works like the MS-DOS command RMDIR.

The pathname must be empty of any files except the working directory('.') and the parent directory('..') Otherwise a "Path not found" or a "Path/File Access error" occurs.

### Example

```
RMDIR "SALES"
```

In this statement, the SALES directory on the current drive is to be removed.

## RND Function

Returns a random number between zero and one.

### Syntax

RND[(X)]

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (see RANDOMIZE Statement).

If  $X > 0$  or  $X$  omitted generates the next random number in the sequence,  $X = 0$  repeats the last number generated.

### Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
```

might return

```
24    30    31    51    5
```

### Note

The values produced by the RND function may vary according to implementation.

## RUN Statement/Command

Runs the program currently in memory, or loads a file into memory and runs it.

### Syntax

RUN [<line number>]  
RUN <filespec> [,R]

With a program in memory running begins either at a specified line or at the lowest line number if a line number is omitted. GW-BASIC always returns to command level after a RUN statement.

For running a program not in memory, the filespec is an optional drive specification followed by a conforming filename. BASIC adds the default filename extension .BAS if no extension is specified.

RUN closes all open files and deletes the current contents of memory before loading the designated program. With the R option, all data files remain open.

### Example

```
RUN "NEWFIL",R
```

## SAVE Command

Saves a program file.

### Syntax

```
SAVE <filespec> [{,A|P}]
```

For saving a program in memory, the filespec is an optional drive specification followed by a conforming filename or pathname. BASIC adds the default filename extension ".BAS" if no extension is specified.

The A option saves the file in ASCII format. If not specified, GW-BASIC saves the file in a compressed binary format. Although ASCII format takes more space on the disk it is needed for actions such as MERGE and for some operating system commands such as TYPE.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOAded), any attempt to list or edit it fails.

### Example

```
SAVE "COM2",A
```

Saves the program COM2 in ASCII format.

```
SAVE "PROG",P
```

Saves the program PROG.BAS as a protected file which cannot be altered.

## SCREEN Statement

Sets the specifications for the display screen. These may vary according to individual machines.

### Syntax

**SCREEN** [*mode*] [, [*colorswitch*]] [, [*apage*]] [, [*vpage*]]

The SCREEN statement is chiefly used to select a screen mode appropriate to a particular display hardware configuration. Supported hardware configurations and screen modes are described below.

### ***Monochrome Display: Mode 0.***

This configuration supports text programs only.

### ***Color Display: Modes 0, 1, and 2.***

This configuration permits running of text mode programs, and both medium resolution and high resolution graphics programs.

### ***EGA with Enhanced Display: Mode 9.***

The full capability of the Enhanced Display is taken advantage of in this mode. The highest resolution possible in any of the modes is with this hardware configuration. Programs written for this mode will not work for any other hardware configuration.

### ***EGA with Monochrome Display: Mode 10.***

The Monochrome Display can be used to display monochrome graphics at a very high resolution in this mode. Programs written for this mode will not work for any other hardware configuration.

### Arguments

The *mode* argument is an integer expression with legal values 0, 1, 2, 9, 10. All other values are illegal. Your selection of a mode argument depends primarily on your program's anticipated display hardware, as described above.

## Colorswitch

For composite monitors and TVs, the *colorswitch* is a numeric expression that is either true (non-zero) or false (zero). A value of zero disables color and permits display of black and white images only. A non-zero value permits color. The meaning of the *colorswitch* argument is inverted in SCREEN mode 0.

For hardware configurations that include an EGA and enough memory to support multiple screen pages, two arguments are available. These *apage* and *vpage* arguments determine the "active" and "visual" memory pages. The active page is the area in memory where graphics statements are written; the visual page is the area of memory that is displayed on the screen. Animation can be achieved by alternating display of graphics pages. The goal here is to display already completed graphics output on the visual page, while executing graphics statements in one or more active pages. A page is displayed only when graphics output to that page is complete. Thus the following is typical:

```
SCREEN 7,,1,2  'work in page 1, show page 2
```

```
.  Graphics output to page 1  
.  while viewing page 2
```

```
SCREEN 7,,2,1  'work in page 2, show page 1
```

```
.  Graphics output to page 2  
.  while viewing page 1  
.
```

The number of pages available depends on the SCREEN mode and the amount of available memory, as described in the following table.

## SCREEN Mode Specifications

Mode	Resolution	Attribute Range	Color Range	EGA Memory	Pages	Page Size
0	40-column text	NA	0-15*	NA	1	2K
	80-column text	NA	0-15*	NA	1	4K
1	320 x 200	0-3†	0-3	NA	1	16K
2	640 x 200	0-1†	0-1	NA	1	16K
7	320 x 200	0-15	0-15	64K	2	32K
				128K	4	
				256K	8	
8	640 x 200	0-15	0-15	64K	1	64K
				128K	2	
				256K	4	
9	640 x 350	0-3	0-15	64K	1	64K
		0-15	0-63	128K	1	128K
		0-15	0-63	256K	2	
10	640 x 350	0-3	0-8	128K	1	128K
				256K	2	

† Attributes applicable only with EGA

\* Numbers in the range 16-31 are blinking versions of the colors 0-15.

## Attributes and Colors

For various screen modes and display hardware configurations, different attribute and color settings exist. (See the "PALETTE Statement" for a discussion of attribute and color number.) The majority of these attribute and color configurations are summarized in the following table.

### Default Attributes and Colors For Most Screen Modes

Attribute Value for Mode			Color Display ‡ Color		Monochrome Display ‡ Pseudo-color	
1,9	2	0,7,8,9†				
0	0	0	0	Black	0	Off
		1	1	Blue		(Underlined) (*)
		2	2	Green	1	On (*)
		3	3	Cyan	1	On (*)
		4	4	Red	1	On (*)
		5	5	Magenta	1	On (*)
		6	6	Brown	1	On (*)
		7	7	White	1	On (*)
		8	8	Gray	0	Off
		9	9	Light Blue		High intensity (underlined)
1		10	10	Light Green	2	High intensity
		11	11	Light Cyan	2	High intensity
		12	12	Light Red	2	High intensity
2		13	13	Light Magenta	2	High intensity
		14	14	Yellow	2	High intensity
3	1	15	15	High-intensity White	0	Off

\* Off when used for background

† With EGA memory > 64K

‡ Only for mode 0 monochrome

## SCREEN Function

Reads a character or its colour from a specified screen location.

### Syntax

SCREEN (<row>,<column>[,z])

<row> is a valid numeric expression returning an unsigned integer.

<column> is a valid numeric expression returning an unsigned integer.

[,z] is a valid numeric expression.

The ordinate of the character at the specified coordinates is stored in the numeric variable.

If the optional parameter z is given and is non-zero, the colour attribute for the character is returned instead.

### Example

```
100 x = SCREEN (10,10)
```

If the character at (10,10) is A, then the function returns 65, the ASCII code for A.

### Example

```
100 x = SCREEN (1,1,1)
```

Returns the colour attribute of the character in the upper left corner of the screen.

## SGN Function

Indicates the value of X, relative to zero.

### Syntax

SGN(X)

If  $X > 0$ , SGN(X) returns 1.

If  $X = 0$ , SGN(X) returns 0.

If  $X < 0$ , SGN(X) returns -1.

### Example

```
ON SGN(X) + 2 GOTO 100,200,300
```

Branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

## SHELL Statement

Exits from the BASIC program to run a .COM .EXE or built in DOS function. Returns to the BASIC program at the line after the SHELL Statement.

### Syntax

SHELL [ <command-string> ]

A COM, EXE or BAT program or DOS function which runs under the SHELL statement is called a "Child Process".

Child processes are executed by SHELL loading and running a copy of command with the "/C" switch. By using COMMAND in this way command line parameters are passed to the child.

Standard input and output may be redirected, and built in commands such as DIR, PATH and TYPE may be executed.

The <command string> must be a valid string expression containing the name of a program to run and (optionally) command arguments.

The program name in <command string> may have any extension you wish. If no extension is supplied, COMMAND will look for a .COM file then a .EXE file and finally a .BAT file.

If COMMAND is not found, SHELL will issue a "File not found" error. No error is generated if COMMAND cannot find the command specified in <command string>.

SHELL with no <command string> will give you a new COMMAND shell. You may now do anything that COMMAND allows. When ready to return to BASIC enter the DOS command: EXIT.

## SIN Function

Returns the sine of X, where X is in radians.

### Syntax

SIN(X)  
COS(X) = SIN(X + 3.14159/2).

### Example

```
PRINT SIN(1.5)
```

returns

```
.9974951
```

See also COS Function.

## SOUND Statement

Generates a sound through the speaker.

### Syntax

SOUND <freq>, <duration>

<freq is in hertz>. This must be a numeric expression returning an unsigned integer.

<duration> is the time in clock ticks. These occur 18.2 times per second. Duration must be a numeric expression returning an unsigned integer in the range zero to 65535. If zero, any current SOUND statement running is turned off.

If no SOUND statement is running a SOUND statement with a duration of zero has no effect.

## Example

```
30 SOUND RND*1000 + 37,2
```

This creates random sounds.

## SPACE\$ Function

Returns a string of spaces of length I.

### Syntax

SPACE\$(I)

The expression I is rounded to an integer. It must be in the range zero to 255.

## Example

```
10 FOR I = 1 TO 5  
20 X$ = SPACE$(I)  
30 PRINT X$;I  
40 NEXT I
```

returns

```
1  
 2  
  3  
   4  
    5
```

See also SPC Function.

## SPC Function

Inserts spaces in a PRINT statement.

### Syntax

SPC(n)

(n) is the number of spaces to be inserted. It must be the range zero to 255.

SPC can only be used with PRINT and LPRINT. A ';' is assumed to follow the SPC(n) command.

### Example

```
PRINT "OVER" SPC(15) "THERE"
```

returns

```
OVER THERE
```

Also see SPACE\$ Function.

## SQR Function

Returns the square root of X.

### Syntax

SQR(X)

X must be  $\geq 0$ .

### Example

```
10 FOR X=10 TO 25 STEP 5  
20 PRINT X, SQR(X)  
30 NEXT X
```

returns

```
10 3.162278  
15 3.872984  
20 4.472136  
25 5
```

## STICK Function

Returns the x and y coordinates of the two joysticks.

### Syntax

STICK(n)

x is a numeric variable for storing the result of the function.

(n) is a numeric expression returning an unsigned integer in the range zero to three:

- \* 0 - returns the x coordinate for joystick A.  
Also stores the x and y values for both joysticks for the following function calls:
- \* 1 - returns the y coordinate of joystick A.
- \* 2 - returns the x coordinate of joystick B.
- \* 3 - returns the y coordinate of joystick B.

### Example

```
10 CLS
20 LOCATE 1,1
30 PRINT "X=5";STICK(0)
40 PRINT "Y=5";STICK(1)
50 GOTO 20
```

This creates an endless loop to display the value of the x,y coordinate for joystick A.

## STOP Statement

Ends a program and returns to command level.

### Syntax

STOP

The STOP Statement doesn't close files. It can be used anywhere in a program and is often used for debugging. When a STOP is met the message "Break in line nnnnn" is printed.

GW-BASIC always returns to command level after a STOP. The program is resumed through a CONT command.

### Example

```
10 INPUT A,B,C
20 K=A ^2*5.3:L= B ^3/.26
30 STOP
40 M=C*K+100:PRINT M
```

returns

```
? 1,2,3
BREAK IN 30
PRINT L
30.76923
CONT
115.9
```

## STR\$ Function

Returns a string representation of the value of n.

### Syntax

STR\$(n)

### Example

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN (STR$ (N)) GOSUB 30, 100, 200, 300,
400, 500
.
.
.
```

See also VAL Function.

## STRIG Function

Returns the status of a specified joystick trigger.

### Syntax

STRIG(n)

where x is a numeric variable for storing the result of the function.

(n) is a numeric expression returning an unsigned integer in the range zero to three designating the trigger to be checked:

- \* 0 - Returns -1 if trigger A has been pressed since the last STRIG(0) statement; returns 0 if not.
- \* 1 - Returns -1 if trigger A is currently down, 0 if not.
- \* 2 - Returns -1 if trigger B has been pressed since the last STRIG(2) statement, 0 if not.
- \* 3 - Returns -1 if trigger B is currently down, 0 if not.

A joystick event trap is not remembered. As a result the `x = STRIG(n)` function always returns false inside a subroutine unless the event has been repeated since the trap. To perform different procedures for various joysticks set up a different one for each rather than include all the procedures in a single routine.

### Example

```
10 IF STRIG(0) THEN BEEP
20 GOTO 10
```

An endless loop is created to beep whenever the trigger button on joystick 0 is pressed.

## STRIG ON, STRIG OFF, STRIG STOP Statements

STRIG ON enables event trapping of joystick activity.  
STRIG OFF disables event trapping of joystick activity.  
STRIG STOP also disables event trapping of joystick activity.

### Syntax

STRIG ON  
STRIG OFF  
STRIG STOP

The STRIG ON statement enables joystick event trapping by an ON STRIG statement. While trapping is enabled, and if a non-zero line number is specified in the ON STRIG statement, GW-BASIC checks between every statement to see if the joystick trigger has been pressed.

The STRIG OFF statement disables event trapping. Any subsequent event is not remembered when the next STRIG ON is invoked. The STRIG STOP statement disables event trapping. However, the event is remembered and a trap takes place when re-enabled.

## STRING\$ Function

Returns a string of length I whose characters have ASCII code J or the first character of X\$.

### Syntax

STRING\$(I,J)  
STRING\$(I,X\$)

### Example

```
10 DASH$ = STRING$(10,45)  
20 PRINT DASH$;"MONTHLY REPORT";DASH$
```

returns

```
-----MONTHLY REPORT-----
```

```
10 LET A$ = "HARRY"  
20 LET X$ = STRING$(8,A$)  
30 PRINT X$
```

returns

```
HHHHHHHH
```

## SWAP Statement

Exchanges the values of two variables.

### Syntax

SWAP <variable> , <variable>

Any type variables can be swapped but they must be the same type to avoid a "Type mismatch" error.

The second variable should already be defined when SWAP is performed otherwise there will be an "Illegal function call".

### Example

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"  
20 PRINT A$ C$ B$  
30 SWAP A$, B$  
40 PRINT A$ C$ B$
```

returns

```
ONE FOR ALL  
ALL FOR ONE
```

## SYSTEM Command

Closes all open files, returns control to the operating system and exits.

### Syntax

SYSTEM

## TAB Function

Moves the print position to I.

### Syntax

TAB(I)

If the current print position is already beyond space I, TAB goes to that position on the next line. Space one is the leftmost position, and the rightmost position is the width minus one. I must be in the range one to 255. You can only use PRINT and LPRINT statements.

### Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
```

returns

NAME	AMOUNT
G. T. JONES	\$25.00

## TAN Function

Returns the tangent of X. X should be given in radians.

### Syntax

TAN(X)

With the interpreter, if TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied and the program continues.

### Example

```
10 Y = Q*TAN(X)/2
```

## TIME\$ Statement

Sets the time; complements the TIME\$ function which retrieves the time.

### Syntax

TIME\$ = <string expression>

<string expression> returns a string in one of the following forms:

hh (sets the hour; minutes and seconds default to 00)

hh:mm (sets the hour and minutes; seconds default to 00)

hh:mm:ss (sets the hour, minutes, and seconds)

A 24-hour clock is used; 8:00 p.m. is entered as 20:00:00.

### Example

```
10 TIME$ = "08:00:00"
```

The current time is set at 8:00 a.m.

## TIME\$ Function

Retrieves the current time.

### Syntax

TIME\$

The TIME\$ function returns an eight-character string in the form hh:mm:ss. hh is the hour (00 through 23), mm is minutes (00 through 59), and ss is seconds (00 through 59). A 24-hour clock is used; 8:00 p.m. is shown as 20:00:00.

### Example

```
10 PRINT TIME$
```

Prints the time, taken from the internal clock.

## TIMER ON, TIMER OFF, TIMER STOP Statements

TIMER ON enables event trapping during real time.  
TIMER OFF disables event trapping during real time.  
TIMER STOP suspends real time event trapping.

### Syntax

TIMER ON  
TIMER OFF  
TIMER STOP

The TIMER ON statement enables real time event trapping by an ON TIMER statement. With the ON TIMER statement, GW-BASIC checks between every statement to see if the timer has reached the specified level. If it has, the ON TIMER statement is performed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent TIMER ON is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an ON TIMER statement performed as soon as trapping is enabled.

See also the ON TIMER Statement.

## TRON/TROFF Statements/Commands

Traces program statements.

### Syntax

TRON  
TROFF

Use TRON as an aid in debugging either direct or indirect mode. Each line number of the program is printed on the screen as it is executed.

The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is effected).

## Example

```
TRON
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K + 10
60 NEXT J
70 END
```

returns

```
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
```

## UNLOCK Statement

Syntax

UNLOCK[#]n[,[( <record number> )][(TO <record number> )]]

The UNLOCK statement is the mirror image to the LOCK statement. UNLOCK releases LOCKS applied to an opened file.

In the case of files opened in random mode, if a range of record numbers is specified, this range must match exactly the record number range given by the LOCK statement.

See LOCK Statement.

## USR Function

Calls an assembly language subroutine.

### Syntax

USR[ <digit> ][( <argument> )]

<digit> specifies which USR routine is being called. See the DEF USR statement for rules governing digit. If omitted, USR0 is assumed.

<argument> is the value passed to the subroutine. It can be any numeric or string expression.

A DEF SEG statement must be used before a USR function call for any segment other than the default segment (data segment). The address given in the DEF SEG statement determines the segment address of the subroutine.

For each USR function, a corresponding DEF USR statement must be used to define the USR call offset. This offset and the current DEF SEG segment address determine the starting address of the subroutine.

### Example

```
100 DEF SEG = &H8000
110 DEF USR0 = 0
120 X = 5
130 Y = USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the value must be consistent with the argument passed. This setup of the string space differs from that of the interpreter.

## VAL Function

Returns the numeric value of string . Also strips leading blanks, tabs, and linefeeds from the argument string.

### Syntax

VAL(<string> )

<string> must be a numeric character stored as such.

### Example

```
VAL(" - 3")
```

returns - 3.

### Example

```
10 READ NAME$,AGE$  
20 IF VAL(AGE$) < 21 THEN PRINT NAME$;"IS TOO  
YOUNG"
```

See the STR\$ function for details on numeric-to-string conversion.

## VARPTR Function

Returns the address of the first byte of data identified with the variable name. For sequential files, returns the starting address of the disk I/O buffer assigned to file number. For random files, returns the address of the FIELD buffer assigned to < file number > .

### Syntax

`VARPTR(<variable name>) VARPTR(#<file number>)`

The variables must have been defined, by carrying out any reference to them, prior to the VARPTR function or an "Illegal function call" results.

Numeric, string or array variable names can be used. For string variables, the address of the first byte of the string descriptor is returned.

The address returned is an integer in the range 32767 to - 32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR gets the address of a variable or array so it can be passed to an assembly language subroutine. A function call of the form `VARPTR(A(0))` is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

All simple variables should be assigned before calling VARPTR as the addresses of the arrays will change.

### Example

```
100 X=USR(VARPTR(Y))
```

## VARPTR\$ Function

Returns a character form of the memory address of the variable in a compatible form for any later programs.

### Syntax

VARPTR\$( <variable name> )

<variable name> is the name of a variable in the program.

VARPTR\$ is mainly used for substrings with the DRAW and PLAY statements in programs not yet compiled. The standard syntax of the PLAY and DRAW statements produces the desired effects.

The variable must have been defined, through carrying out any reference to it, before the VARPTR function or an "Illegal function call" results.

VARPTR\$ returns a three-byte string in the form:

byte 0 = type

byte 1 = low byte of address

byte 2 = high byte of address

The individual parts of the string are not considered characters.

### Note

Do not save the result of a VARPTR function in a variable as array and string addresses and file data block change whenever a new variable is assigned. VARPTR should be executed before each use of the result.

### Example

```
10 PLAY "X" + VARPTR$(A$)
```

Uses the subcommand X (execute), plus the contents of A\$, as the string expression in the PLAY statement.

## VIEW Statement

Defines screen limits for graphics activity.

### Syntax

```
VIEW [[SCREEN] [(Vx1,Vy1)-(Vx2,Vy2) [, [<colour> ]  
[, [<border> ]]]]]
```

VIEW defines a "Physical Viewport" limit from Vx1,Vy1 (upper left x,y coordinates) to Vx2,Vy2 (lower right x,y coordinates). The x and y coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which graphics may be mapped.

RUN, and RUN, SCREEN and VIEW with no arguments, define the entire screen as the viewport.

With colour the view area can be filled. Border lets a line be drawn surrounding the viewport if space for a border is available.

With the SCREEN option the x and y coordinates are absolute to the screen, not relative to the border of the physical viewport. Only graphics within the viewport are plotted.

### Example

```
VIEW (Vx1,Vy1)-(Vx2,Vy2)
```

All points plotted are relative to the viewport. That is, Vx1 and Vy1 are added to the x and y coordinates before putting the point down on the screen.

```
VIEW (10,10)-(200,100)
```

If carried out, the point set down by the statement PSET (0,0),3 would be at the physical screen location 10,10.

```
VIEW SCREEN (Vx1,Vy1)-(Vx2,Vy2)
```

All coordinates are screen absolute rather than viewport relative.

```
VIEW SCREEN (10,10)-(200,100)
```

If carried out, the point set down by the statement PSET (0,0),3 would not appear because 0,0 is outside the Viewport. PSET (10,10),3 is within the Viewport, and places the point in the upper-left hand corner.

There can be a number of VIEW statements. If the newly described viewport is not entirely within the previous viewport, the screen can be re-initialised with the VIEW statement and the new viewport then stated. The intermediate VIEW statement is not necessary if the new viewport is wholly within the previous one.

The following example opens three viewports, each smaller than the previous one. In each case, a line is defined to go beyond the borders but it appears only within the viewport border.

```
260 CLS
280 VIEW: REM ** Make the viewport the entire screen
300 VIEW (10,10) - (300,180),,1
320 CLS
340 LINE (0,0) - (310,190),1
360 LOCATE 1,11: PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400 CLS:REM** Note, CLS clears only viewport.
420 LINE (300,0)-(0,199),1
440 LOCATE 9,9: PRINT "A medium viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480 CLS
500 CIRCLE (150,100),20,1
520 LOCATE 11,9: PRINT "A small viewport"
```

## VIEW PRINT Statement

Sets the boundaries of the screen text window.

### Syntax

VIEW PRINT [ < top screen line > TO < bottom screen line > ]

VIEW PRINT without top and bottom line parameters initialises the whole screen area as the text window.

Statements and functions operating within the defined text window include CLS, LOCATE PRINT and SCREEN.

The Screen Editor limits functions such as scroll and cursor movement to the text window. Also see the VIEW Statement.

## WAIT Statement

Suspends a program while monitoring the status of a machine input port.

### Syntax

WAIT < port number > ,I[,J]

I and J are integer expressions.

With the WAIT statement the program is suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusively OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is nonzero, the program continues with the next statement. If J is omitted, it is assumed to be zero.

### Warning

It is possible to enter an infinite loop with the WAIT statement. You then have to manually restart the machine. To avoid this, WAIT must have the specified value at port number during some point in the program.

### Example

```
100 WAIT 32,2
```

## WHILE...WEND Statements

Performs a series of statements in a loop as long as a given condition is true.

### Syntax

WHILE <expression>

.

.

[ <loop statements> ]

.

WEND

If <expression> is not zero ('true') loop statements are performed until the WEND statement is met. GW-BASIC then returns to the WHILE statement and checks the expression. If it is still true, the process is repeated. If it is not true, the program resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement a "WEND without WHILE" error.

### Example

```
90 'BUBBLE SORT ARRAY A$ WHICH HAS J
ELEMENTS.
100 FLIPS=1 'FORCE ONE PASS THROUGH LOOP
110 WHILE FLIPS
115 FLIPS=0
120 FOR I=1 TO J-1
130 A$(I)>A$(I+1) THEN SWAP A$(I),A$(I+1):
FLIPS=1
140 NEXT I
150 WEND
```

### Note

Do not direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

## WIDTH Statement

Sets the printed line width in number of characters for the screen or line printer.

### Syntax

```
WIDTH [LPRINT] <size>  
WIDTH <file number> , <size>  
WIDTH <device> , <size>
```

<size> is a numeric expression in the range zero to 255. It specifies the width of the printed line. The default width is 72 characters.

If integer expression is 255, the line width is "infinite"; and a carriage return is never used. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

<file number> is a numeric expression in the range one to 15 and is the number of the opened file; <device> is a string expression indicating the device to be used.

WIDTH can clear the screen.

If the LPRINT option is omitted, the line width is set for the screen; if included it is set for the line printer.

WIDTH <file number> , <size> immediately changes the width of any open file to the specified size.

With WIDTH <device> <size> the default line width for the specified device is set to size. The line widths of currently open files are not modified. A subsequent OPEN filespec FOR OUTPUT AS #n uses the specified value for the width initially.

## Example

```
10 WIDTH "LPT1:", 5
20 OPEN "LPT1:" FOR OUTPUT AS 1
30 PRINT 1, "1234567890"
35 PRINT 1
40 WIDTH 1, 6
50 PRINT 1, "1234567890"
RUN
```

returns on the printer

```
12345
67890
123456
7890
```

## WINDOW Statement

Defines the logical dimensions of the current viewport.

### Syntax

**WINDOW** [[**SCREEN**] (Wx1,Wy1)-(Wx2,Wy2)]

(Wx1,Wy1)-(Wx2,Wy2) are the world coordinates defining the coordinates of the lower left and upper right screen border.

**SCREEN** inverts the y axis of the world coordinates so that screen coordinates coincide with the traditional Cartesian arrangement; x increases left to right, and y decreases top to bottom.

With **WINDOW** the screen border coordinates can be redefined. Lines, graphs or objects can be drawn in space not bounded by the physical dimensions of the screen. This is done by using programmer-defined world coordinates. When the screen is redefined, graphics can be drawn within a customised mapping system.

BASIC converts world coordinates into physical coordinates for subsequent display within the current viewport. To make this transformation the portion of the (floating point) world coordinate space containing the information to be displayed must be known. This rectangular region in world coordinate space is called a Window.

RUN, or WINDOW with no arguments, disables Window transformation.

The WINDOW SCREEN variant inverts the normal Cartesian direction of the y coordinate. In the default, a section of the screen appears as:

0,0	50,0	100,0
↓		
	increases	
	100,0	
0,100	50,100	100,100

now carry out

WINDOW (-1,-1)-(1,1)

and the screen appears as:

-1,1	0,1	1,1
↑ y increases		
	0,0	
↓ y decreases		
-1,-1	0,-1	1,-1

If the variant:

WINDOW SCREEN (-1,-1)-(1,1)

is carried out then the screen appears as:

-1,-1	0,-1	1,-1
↑ y decreases		
	0,0	
↓ y increases		
-1,1	0,1	1,1

The following example illustrates two lines with the same endpoint coordinates. The first is drawn on the default screen, and the second on a redefined window.

```
200 LINE (100,100) - (150,150), 1
220 LOCATE 2,20:PRINT "The line on the default
screen"
240 WINDOW SCREEN (100,100) - (200,200)
260 LINE (100,100) - (150,150), 1
280 LOCATE 8,18:PRINT "& the same line on a
redefined window"
```

## WRITE Statement

Outputs data to the screen.

### Syntax

WRITE [ <list of expressions> ]

If <list of expressions> is omitted, a blank line is output. If included, the values of the expressions are output to the screen. The expressions list may be numeric or string. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, GW-BASIC inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement.

### Example

```
10 A=80:B=90:C$="THAT'S ALL"  
20 WRITE A,B,C$
```

returns

```
80, 90, "THAT'S ALL"
```

## WRITE# Statement

Writes data to a sequential file. This statement should be used rather than PRINT# if the information is to be read back by a program.

### Syntax

WRITE# <file number> , <list of expressions>

File number is the number under which the file was OPENed in O mode (see OPEN Statement). The expressions in the list are string or numeric. They must be separated by commas.

WRITE#, unlike PRINT , inserts commas between the items as they are written to the file and delimits strings with quotation marks. It is not necessary to put explicit delimiters in the list. GW-BASIC inserts a return/linefeed sequence after the last item in the list is written to the file.

### Example

Let A\$ = "CAMERA" and B\$ = "93604 - 1"  
The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604 - 1"
```

A subsequent statement such as

```
INPUT#1,A$,B$
```

would input "CAMERA" to A\$ and "93604-1" to B\$.

# ASSEMBLY LANGUAGE SUBROUTINES



This section assumes a knowledge of machine language.

Call assembly language subroutines from your GW-BASIC program with the USR function or the CALL or CALLS statement.

We recommend you use CALL or CALLS for interfacing 8086 machine language programs with GW-BASIC. These statements are more readable and can pass multiple arguments. In addition, the CALL statement is compatible with more languages than USR.

## Memory Allocation

Use the /M: switch during start-up to set aside memory space for assembly language subroutines.

/M sets the highest memory location to be used by GW-BASIC.

In addition to its code area GW-BASIC uses up to 64K of memory beginning at its data (DS) segment.

If more stack space is needed when an assembly language subroutine is called, save the GW-BASIC stack and set up a new stack for use by the assembly language subroutine.

Restore the GW-BASIC stack before returning from the subroutine.

The BLOAD command (see Section 5) is the simplest way to load assembly language subroutines into memory.

Alternatively you can run a program that exists but stays resident, and then run BASIC.

If you BLOAD, or read and poke, an EXE file into memory:

- \* Make sure the subroutines do not contain any long references - address offsets which exceed 64k or will be out of the code segment. Long references need handling by the EXE loader.
- \* Skip over the first 512 bytes (the header) of the linker's output file (EXE), then read in the rest of the file.

## Internal Representation of numbers

For many assembly language programming routines you need to know the internal representation of numbers in GW-BASIC.

### Single Precision - 24 bit mantissa



loman = the low mantissa

S = the sign

himan = the high mantissa

exp = the exponent

man = himan:...:loman

- \* If <exp> equals 0, then <number> equals 0.
- \* If <exp> is not equal to 0, then the mantissa is normalised and <number> equals <sgn> \*.1 <man> \* 2 \*\* (<exp> - 80h)

That is, in single precision (hex notation - bytes low to high)

00000080 = .5

00008080 = -.5

### Double Precision - 56 bit mantissa



## CALL Statement

Use CALL to interface 8086 machine language subroutines with GW-BASIC 3.1. Do not use the USR function unless you are running previously written subroutines already containing USR functions.

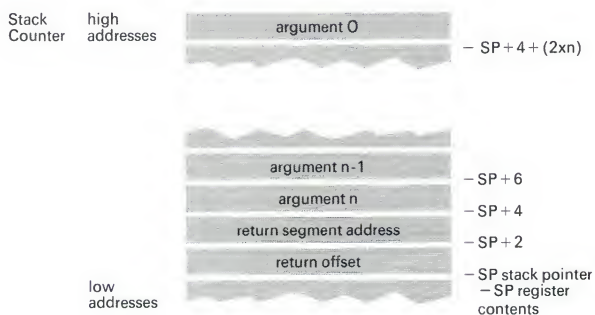
Syntax is:

CALL <variable name> [( <argument list> )] where <variable name> contains the offset into the current segment for the starting point in memory of the called subroutine. The current segment is either the default, or defined by a DEF SEG statement.

<argument list> contains the variables or constants, separated by commas, to be passed to the subroutine.

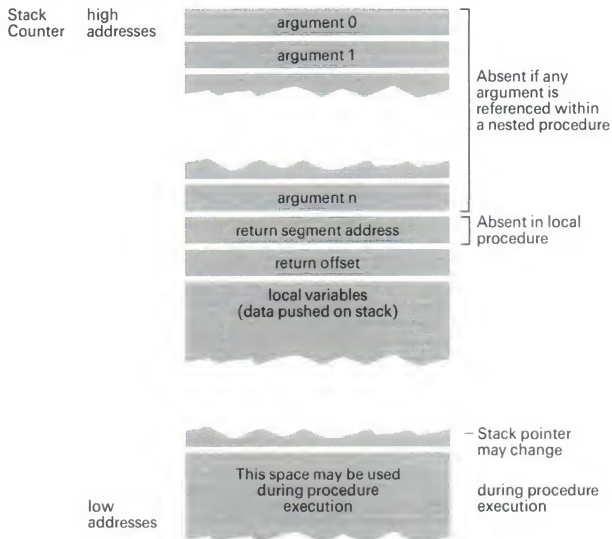
Using CALL means that:

- \* For each argument listed, the two-byte offset of the argument's location within the BASIC segment is pushed onto the stack.
- \* Control is transferred to the subroutine with an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name> .



Stack layout when CALL is activated.

After CALL has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.



Stack layout during execution of a CALL statement

## Rules for coding a subroutine

- \* The called routine must preserve segment registers DS, ES, SS, and the base pointer (BP).

If interrupts are disabled in the routine, they must be enabled before exiting. The stack must be cleaned up on exit.

- \* The called program must know the number and length of the arguments passed. An easy way to reference arguments is:

```
PUSH    BP
MOV     BP,SP
ADD     BP, (2*number of arguments) + 4
```

Then:

argument 0 is at BP

argument 1 is at BP-2

argument n is at BP-2\*n

(number of arguments = n + 1)

- \* Allocate variables either in the code segment or on the stack. Do not modify the return segment and offset stored on the stack.
- \* The called subroutine must clean up the stack; do this with a RET <n> statement to adjust the stack to the start of the calling sequence. <n> is twice the number of arguments in the argument list.
- \* Values are returned to variables named in the argument list. (See the *Internal Representation of Numbers* section)
- \* If the argument is a string, the argument's offset points to a 3 byte-unit, called the string descriptor. The first byte of the string descriptor contains the length of the string (zero to 255). The second and third are, respectively, the lower and upper eight bits of the string starting address in string space.

## Warning

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. For example

```
20 A$ = "BASIC" + ""
```

This copies the string literal into string space so that it can be modified without affecting the program.

- \* User routines can alter the contents of a string but the descriptor must not be changed. Do not write past the end-of-string. GW-BASIC 3.1 cannot correctly manipulate strings if their lengths are modified by external routines.
- \* Allocate data areas needed by the routine either in the CODE segment of the user routine or on the stack. You cannot declare a separate data area in the user assembler routine.

## Example CALL statement

```
100 DEF SEG = &H8000
110 FOO = &H7FA
120 CALL FOO(A,B$,C)
```

Line 100 sets the segment to 8000 Hex. The value of variable FOO is added into the address as the low word after the DEF SEG value is left shifted 4 bits. Here, the long call to FOO will carry out the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence in 8086 assembly language demonstrates access to the arguments passed. The returned result is stored in the variable 'C'.

PUSH	BP	;Set up pointer to arguments
MOV	BP,SP	
ADD	BP,(4 + 2*3)	
MOV	BX,[BP-2]	;Get address of B\$ descriptor.
MOV	CL,[BX]	;Get length of B\$ in CL.
See MOV	DX,1[BX]	;Get addr of B\$ text in DX.
.		
.		
.		
MOV	SI,[BP]	;Get address of 'A' in SI.
MOV	DI[BP-4]	;Get pointer to 'C' in DI.
MOVS	WORD	;Store variable 'A' in 'C'.
POP	BP	;Restore pointer.
RET	6	;Restore stack, return.

## Warning

The called program must know the variable type for the numeric arguments passed. In the previous example, the instruction `MOVS WORD` will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy eight bytes if they were double precision.

## CALLS Statement

Use `CALLS` to access subroutines written using MS-FORTRAN calling conventions. `CALLS` works just like `CALL`, but with `CALLS` arguments are passed as segmented rather than unsegmented addresses.

MS-FORTRAN routines need to know the segment value for each argument passed, so both the segment and the offset are pushed. For each argument, four bytes are pushed rather than 2 as with `CALL`. If your assembler routine uses `CALLS`, in the `RET <n>` statement `<n>` is four times the number of arguments.

## USR Function

CALL is the recommended way to call assembly language subroutines, but you can also use the USR function. This ensures compatibility with older programs containing USR functions.

Syntax is:

USR[<digit>][(<argument>)]

<digit> is from zero to nine and specifies the USR routine being called. If omitted, USR0 is assumed.

<argument> is any numeric or string expression.

To ensure that the code segment points to the subroutine being called, you must include a DEF SEG statement before a USR function call. The segment address in the DEF SEG statement determines the starting segment of the subroutine.

For each USR function, there must be a DEF USR statement to define the USR function call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value specifying the type of argument given. The value may be:

Value in AL	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

## Number arguments

The BX register points to Floating-Point Accumulator (FAC) where the argument is stored.

For integer arguments:

FAC-2 contains the upper eight bits of the integer.

FAC-3 contains the lower eight bits of the integer.

For versions of GW-BASIC which use binary floating-point:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest seven bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative).

For single precision floating point numbers:

FAC-2 contains the middle eight bits of mantissa.

FAC-3 contains the lowest eight bits of mantissa.

For double precision floating-point numbers:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest eight bits).

## String arguments

The DX register points to a 3-byte unit called the string descriptor. The first byte holds the length of the string (zero to 255 characters).

The second and third bytes are, respectively, the lower and upper eight bits of the string starting address in the GW-BASIC data segment.

### Warning

If the argument is a string literal in the program, the string descriptor will point to program text. Do not alter or destroy the program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument passed to it.

In GW-BASIC 3.1, USR allows calls to MAKINT and FRCINT giving access to the routines without having to supply their absolute addresses. The address ES:BP is used as an indirect far pointer to the routines FRCINT and MAKINT.

To call FRCINT from a USR routine use `CALL DWORD ES:[BP]`.

To call MAKINT from a USR routine use `CALL DWORD ES:[BP + 4]`.

### Example

```
110 DEF USR0 = &H8000 'Assumes decimal  
argument /M:32767  
120 X = 5  
130 Y = USR0(X)  
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument used.

# APPENDICES





# 7.1 ASCII Character codes

Dec = decimal, Hex = hexadecimal (H), CHR = character.  
 LF = Line Feed, FF = Form Feed, CR = Carriage Return,  
 DEL = Rubout

<b>Dec</b>	<b>Hex</b>	<b>CHR</b>	<b>Dec</b>	<b>Hex</b>	<b>CHR</b>
000	00H	NUL	033	21H	!
001	01H	SOH	034	22H	"
002	02H	STX	035	23H	#
003	03H	ETX	036	24H	\$
004	04H	EOT	037	25H	%
005	05H	ENQ	038	26H	&
006	06H	ACK	039	27H	,
007	07H	BEL	040	28H	(
008	08H	BS	041	29H	)
009	09H	HT	042	2AH	*
010	0AH	LF	043	2BH	+
011	0BH	VT	044	2CH	,
012	0CH	FF	045	2DH	-
013	0DH	CR	046	2EH	.
014	0EH	SO	047	2FH	/
015	0FH	SI	048	30H	0
016	10H	DLE	049	31H	1
017	11H	DC1	050	32H	2
018	12H	DC2	051	33H	3
019	13H	DC3	052	34H	4
020	14H	DC4	053	35H	5
021	15H	NAK	054	36H	6
022	16H	SYN	055	37H	7
023	17H	ETB	056	38H	8
024	18H	CAN	057	39H	9
025	19H	EM	058	3AH	:
026	1AH	SUB	059	3BH	;
027	1BH	ESCAPE	060	3CH	<
028	1CH	FS	061	3DH	=
029	1DH	GS	062	3EH	>
030	1EH	RS	063	3FH	?
031	1FH	US	064	40H	@
032	20H	SPACE			

(character codes continued)

Dec	Hex	CHR	Dec	Hex	CHR
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[	123	7BH	{
092	5CH	\	124	7CH	
093	5DH	]	125	7DH	}
094	5EH	^	126	7EH	~
095	5FH	_	128	7FH	DEL
096	60H	'			

# GW-BASIC Error codes and error messages.

## 7.2

### 1 NEXT without FOR

A variable in a NEXT statement does not match any executed unmatched FOR statement variable.

### 2 Syntax error

A line having some incorrect sequence of characters is met; for example, unmatched parenthesis, misspelled command or statement, incorrect punctuation. With GW-BASIC the incorrect line is part of a DATA statement. The Interpreter automatically enters edit mode at the line causing the error.

### 3 Return without GOSUB

A RETURN statement not having a previous, unmatched GOSUB statement is met.

### 4 Out of data

A READ statement is performed when there are no DATA statements with unread data remaining in the program.

### 5 Illegal function call

A parameter out of range is passed to a math or string function. It can also be due to:

- \* A negative or unreasonably large subscript.
- \* A negative or zero argument with LOG.
- \* A negative argument to SQR.
- \* A negative mantissa with a noninteger exponent.
- \* A call to a USR function not having a starting address.
- \* An improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO.
- \* A negative record number used with GET or PUT.

## **6 Overflow**

The result of a calculation is too large to be represented in GW-BASIC number format. If underflow occurs, the result is zero and the program continues without an error.

## **7 Out of memory**

A program is too large, has too many FOR loops or GOSUBs, too many variables; expressions may be too complicated for a file buffer to be allocated.

## **8 Undefined line**

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

## **9 Subscript out of range**

An array element is referenced either with a subscript outside the dimensions of the array or with the wrong number of subscripts.

## **10 Duplicate definition**

Two DIM statements are given for the same array; or a DIM statement given for an array after its default dimension of 10 has already been established.

## **11 Division by zero**

A division by zero is met in an expression; or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division; or positive machine infinity is supplied as the result of the involution and the program continues.

## **12 Illegal direct**

A statement illegal in direct mode is entered this way.

## **13 Type mismatch**

A string variable name is assigned a numeric value or vice versa; a function expecting a numeric argument is given a string argument or vice versa.

## **14 Out of string space**

String variables have made BASIC exceed the amount of free memory remaining. GW-BASIC allocates string space dynamically, until it runs out of memory.

## **15 String too long**

An attempt is made to create a string more than 255 characters long.

## **16 String formula too complex**

A string expression is too long or too complex. Make the expression smaller.

## **17 Can't continue**

An attempt is made to continue a program that

- \* Has halted due to an error.
- \* Has been modified during a break.
- \* Does not exist.

## **18 Undefined user function**

AUSR function is called before the function definition (DEF statement) is given.

## **19 No RESUME**

An error handling routine not containing a RESUME statement is entered.

## **20 RESUME without error**

A RESUME statement is met before an error handling routine is entered.

## **21 Unprintable error**

An error message is not available for this type of error.

## **22 Missing operand**

An expression contains an operator with no operand following it.

## **23 Line buffer overflow**

An attempt has been made to input a line with too many characters.

**24 Device timeout**

The device specified is not available at this time.

**25 Device fault**

An incorrect device designation has been entered.

**26 FOR without NEXT**

A FOR statement was met without a matching NEXT.

**27 Out of paper**

The printer device is out of paper.

**28 Unprintable error**

An error message is not available for this type of error.

**29 WHILE without WEND**

A WHILE statement does not have a matching WEND.

**30 WEND without WHILE**

A WEND statement does not have a matching WHILE.

**31-49 Unprintable error**

An error message is not available for this type of error.

***Disk Errors*****50 Field overflow**

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

**51 Internal error****52 Bad file number**

A statement or command references a file with a number not OPEN or out of the range of file numbers specified at initialisation.

**53 File not found**

A LOAD, KILL, NAME, or OPEN statement/ command references a file not on the current disk.

#### **54 Bad file mode**

An attempt to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to carry out an OPEN statement with a file mode other than I, O, or R.

#### **55 File already open**

A sequential output mode OPEN statement is given for a file already open; or a KILL statement given for a open file.

#### **56 Unprintable error**

An error message is not available for this type of error.

#### **57 Device I/O error**

An I/O error occurred on a disk I/O operation. It is a fatal error as the operating system cannot recover from it.

#### **58 File already exists**

The filename specified in a NAME statement is identical to a filename in use on the disk.

#### **59-60 Unprintable error**

An error message is not available for this type of error.

#### **61 Disk full**

All disk storage space is in use.

#### **62 Input past end**

An INPUT statement is carried out after all the data in the file has been INPUT; or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.

#### **63 Bad record number**

In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equals zero.

#### **64 Bad file name**

An illegal form, such as a filename with too many characters, is used for the filename with a LOAD, SAVE, KILL, or OPEN statement.

## **65 Unprintable error**

An error message is not available for this type of error.

## **66 Direct statement in file**

A direct statement is met while LOADING an ASCII-format file. The LOAD is ended.

## **67 Too many files**

An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

## **68 Device Unavailable**

The device specified is not available at this time.

## **69 Communications buffer overflow**

Not enough space has been reserved for communications I/O.

## **70 Disk write protected**

The disk has a write protect tab intact; or is a disk that cannot be written to.

## **71 Disk not ready**

Could be caused by a number of problems; most likely the disk is not inserted properly.

## **72 Disk media error**

A hardware or disk problem occurred while the disk was being written to or read from. For example, the disk may be damaged or the disk drive not working properly.

## **74 Rename across disks**

An attempt was made to rename a file with a new drive designation. This is not allowed.

## **75 Path/file access error**

During an OPEN, MKDIR, CHDIR, or RMDIR operation the correct Path to File name connection could not be made The operation is not completed.

## **76 Path not Found**

During an OPEN, MKDIR, CHDIR, or RMDIR operation the path specified could not be found. The operation is not completed.

## GW-BASIC Reserved words

## 7.3

---

ABS	FOR	PEEK	WAIT
AND	FRE	PEN	WEND
ASC	GET	PLAY	WHILE
ATN	GOSUB	PMAP	WIDTH
AUTO	HEX\$	POINT	WINDOW
BEEP	IF	POKE	WRITE
BLOAD	IMP	POS	WRITE#
BSAVE	INP	PRESET	XOR
CALL	INPUT	PRINT	
CDBL	INKEY\$	PRINT# USING	
CHAIN	INPUT#	PSET	
CHDIR	INPUT\$	PUT	
CHR\$	INSTR	RANDOMIZE	
CINT	INT	READ	
CIRCLE	IOCTL	REM	
CLEAR	KEY	RENUM	
CLOSE	KILL	RESET	
CLS	LEFT\$	RESTORE	
COLOR	LEN	RESUME	
COM	LET	RIGHT\$	
COMMON	LINE	RMDIR	
CONT	LIST	RND	
COS	LLIST	RSET	
CSNG	LOAD	RUN	
CVD	LOC	SAVE	
CVI	LOCATE	SGN	
DATA	LOF	SHELL	
DATE\$	LOG	SIN	
DEFDBL	LPOS	SOUND	
DEFINT	LPRINT	SPACE	
DEFSNG	LSET	SPC	
DEFSTR	MERGE	SQR	
DEF FN	MID\$	STICK	
DEF USR	MKD\$	STOP	
DELETE	MKI\$	STR\$	
DIM	MKS\$	STRIG	
DRAW	MKDIR	STRING\$	
EDIT	MOD	SWAP	
ELSE	MOTOR	SYSTEM	

---

(Reserved words continued)

---

END	NAME	TAB
ENVIRON	NEW	TAN
EOF	NEXT	THEN
ERASE	NOT	TIME\$
ERDEV	OCT\$	TO
ERL	ON	TROFF
ERR	OPEN	TRON
ERROR	OPEN COM	USING
END	OPTION	USR
EXP	OR	VAL
FIELD	PAINT	VARPTR
FILES	PALETTE	VARPTR\$
FIX	PALETTE USING	VIEW

---

# Mathematical functions not intrinsic to GW-BASIC

## 7.4

### Derived Functions

These can be calculated as follows;

Function	GW-BASIC Equivalent
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X^2 + 1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X^2 + 1)) + 1.5708$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1)) + \text{SGN}(\text{SGN}(X) - 1) * 1.5708$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) * 1.5708$
INVERSE COTANGENT	$\text{ARCCOT}(X) = \text{ATN}(X) + 1.5708$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = (\text{EXP}(X) - \text{EXP}(-X)) / (\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = (\text{EXP}(X) + \text{EXP}(-X)) / (\text{EXP}(X) - \text{EXP}(-X))$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X^2 + 1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X^2 - 1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1 + X)/(1 - X))/2$

### (Derived functions continued)

Function	GW-BASIC Equivalent
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X*X+1)+1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X)*\text{SQR}(X*X+1)+1)/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X+1)/(X-1))/2$
LOG TO BASE10	$\text{TENLOG}(X) = \text{LOG}(X)/\text{LOG}(10)$

# INDEX



# Index

24 hour clock 5/152

## A

ABS 5/2

Absolute value 5/2

Accessing random access files 4/17

Active display pages 1/10

Adding new lines 3/2

Adding to sequential files 4/14

Addition 2/12

Advanced features 2/1

ALT + A-Z 3/5

ALT key 3/5

AND 2/12

Animation 5/125

ANSI 1/1

Arctangent 5/3

Arguments number 6/9

Arguments referencing 6/4

Arguments string 6/10

Arithmetic operators 2/13

Array 5/27

- deletion 5/33

- redimensioning 5/33

- removal 5/33

- subscripts 5/100

- variables 2/11

- space requirements 2/12

ASC 5/2

ASCII 5/2, 5/12

- character codes 7/2

- mode communications 5/33

Assembly language 5/6, 5/25, 5/155, 6/1

ATN 5/3

AUTO 1/9, 5/3

## B

Background 5/16  
BEEP 5/4  
Bell 5/4  
Binary mode communication 5/33  
BLOAD 5/4, 6/1  
Border 5/16  
Branch on value 5/87  
Break 5/18, 5/70  
BSAVE 5/5  
Buffer - random 4/16

## C

CALL 5/6, 6/1, 6/3  
CALLS 5/6, 6/1, 6/7  
CBDL 5/7  
CHAIN 4/10, 5/7  
Changing directory 5/11  
Character set 2/2  
Characters declaration 2/9  
CHDIR 4/6, 5/11  
Child process 5/31, 5/142  
CHR\$ 5/12  
CINT 5/12  
CIRCLE 5/13  
CLEAR 5/13  
CLOSE 5/15  
CLOSE 4/11  
CLS 5/15  
Coding subroutines 6/5  
COLOR 5/16  
Color display 5/137  
Command line option switches 1/4  
Commands  
    ALT-key equivalents 2/5, 3/5  
        commonly used 3/5  
        program file 4/9  
Common variables 5/7, 5/16, 5/17  
Commonly used commands 2/5, 3/5

- Communications 5/16, 5/85, 5/95
  - channel 5/95
  - device status 5/60
  - file 5/33, 5/47, 5/73
  - port 5/16
- Concatenation - string 2/19
- Constants 2/6
  - converting 2/20
  - double precision numeric 2/8
  - fixed-point 2/6
  - floating-point 2/6
  - hex 2/7
  - integer 2/6
  - numeric 2/6
  - octal 2/7
  - single precision numeric 2/8
  - string 2/6
- CONT 5/18, 5/69
- Control
  - characters 2/4
  - key functions 3/4
  - hex codes 3/4
- Controlled loop 5/44
- Converting constants 2/20
- COS 5/18
- Creating sequential files 4/12
- CRSLIN 5/20
- CSNG 5/19
- CTRL-C 5/4, 5/53, 5/69
- CTRL-Z 4/7, 5/70
- Cursor
  - on/off 5/74
  - position 5/20, 5/74, 5/112
  - style 5/74
- CVD 4/18, 5/20
- CVI 4/18, 5/20
- CVS 4/18, 5/20

## D

DATA 5/21

- alignment 5/79

- files 4/11

Date

- format 5/22

- reading 5/22

- setting 5/22

DATE\$ 5/22, 5/22

Debugging programs 1/8

Declaration characters 2/9

DEDUCTION # 2/21

DEF DBL 5/13, 5/24

DEF FN 2/23, 5/13, 5/23

DEF INT 5/24

DEF SEG 5/4, 5/5, 5/25, 5/155, 6/3, 6/8

DEF SNG 5/13, 5/24

DEF STR 5/13, 5/24

DEF USR 5/25, 6/8

Default device 4/2

Defining error codes 5/35

DELETE 1/9, 5/26

Deleting

- files 5/64

- lines 3/2

Descriptor string 6/6

Devices 4/1

- and files 4/1

- default 4/2

- input 5/60

- output 5/59

- status 5/34

Device-independent I/O 4/3

DIM 5/27

Direct mode 1/8

- Directory 4/5
  - creation 5/82
  - deletion 5/134
  - hierarchical 4/5
  - of files 5/42
  - parent 4/6
  - root 4/5
  - working 4/5
- Display
  - coordinates 5/159, 5/164
  - formatting 5/116, 5/145, 5/151
  - line width 5/163
  - of information 5/113
  - pages active 1/10
  - specification 5/137
- Division 2/12
  - by zero 2/15
  - integer 2/14, 2/12
- Dollar sign 2/9
- Double precision 5/7, 5/24
  - floating point constants 2/7
  - numbers 6/2, 6/8
  - numeric constants 2/8
  - value 4/16
  - variable 2/10, 2/21
- DRAW 5/28
- DX register 6/10

## E

- EDIT 1/9, 3/1, 3/3, 3/6, 5/30
- Editing 3/3
- Editor
  - full screen 3/2
  - GW-BASIC 3/1
- EGA display 5/137
- Elipse 5/13
- END 3/6, 5/30
- End of file 5/32, 5/70
- Ending a BASIC session 5/150
- Enhanced graphics display 5/137
- Entering text 3/2
- ENVIRON 5/30
- ENVIRON\$ 5/32
- Environment string 5/32
- EOF 4/7, 4/11, 5/32, 5/70
- EQV 2/12
- ERASE 5/33
- ERDEV 5/34
- ERDEV\$ 5/34
- ERL 5/35
- ERR 5/35
- Error
  - codes 7/5
    - reading 5/35, 5/35
    - extended information 5/38
  - messages 7/5
  - signalling 3/3
  - syntax 3/6
  - trapping 5/86, 5/133
- Event trapping 2/24, 5/62
- EXP 5/37
- Exponentiation 2/12
- Expressions 2/12
- EXTERR 5/38

## F

FAC 6/9

FIELD 5/39

File

- closure 5/15, 5/30, 5/132

- commands program 4/9

- handling 4/9

- input 5/47, 5/56

- length 5/77

- locking 5/75

- number 5/15, 5/56

- opening 5/97

- operations - random 4/19

- protection 5/154

- renaming 5/83

- unlocking 5/75

Filenames 4/4

- specifications of 4/4

FILES 5/42

Files 4/1

- and devices 4/1

- creating random access 4/15

- creating sequential 4/12

- data 4/11

- random 4/14

- random access 4/14

- reading sequential 4/13

- sequential 4/11

- sequential adding to 4/14

Filespec 1/5

FIX 5/43

Fixed-point constants 2/6

Floating point

- accumulator 6/9

- constants 2/6

- division 2/13

- double precision 2/7

FOR...NEXT 5/44

Foreground 5/16

FRE 5/46

Full screen editor 3/2

- Function
  - definition 5/23
  - key assignment 5/61
- Functions
  - intrinsic 2/23
  - user-defined 2/23

## **G**

- Garbage collection 5/46
- GET (graphics) 5/48
- GET (I/O) 5/47
- GOSUB...RETURN 2.24, 5/48
- GOTO 5/49
- Graphics
  - co-ordinates 5/109, 5/123, 5/159
  - color 5/103
  - cursor co-ordina 5/110
  - image color fill 5/101
  - images 5/48
  - macro language 5/28, 5/28, 5/48, 5/67, 5/101 paint, 5/102, 5/112
- GW-BASIC
  - Editor 3/1
    - writing programs using 3/1
  - leaving 1/3
  - starting 1/3

## **H**

- Handling files 4/9
- Hex codes control key functions 3/4
- Hex constants 2/7
- HEX\$ 5/50
- Hexadecimal 2/2, 5/50
- Hierarchical directories 4/5

## I

I/O Device-independent 4/3

IF...GOTO 5/51

IF...THEN 5/51

IF...THEN...ELSE 5/51

Indirect mode 1/8

Information display 5/167

INKEY\$ 5/53, 5/61

INP 2/12, 5/54

INPUT 3/6, 5/54

    carriage return suppress 5/54, 5/69

    question mark suppress 5/54, 5/69

    re-direction 4/7

INPUT # 4/11, 5/56

INPUT\$ 4/11, 5/57

INSTR\$ 5/58

Integer

    constants 2/6

    conversion 5/12, 5/24, 5/59, 5/59

    division 2/12, 2/14

    truncation 5/43

    variables 2/10

Internal representation of numbers 6/2

Intrinsic functions 2/23

IOCTL 5/59

IOCTL\$ 5/60

## J

Joystick 5/92, 5/146, 5/148, 5/149

## K

Key trapping 5/87

KEY(n) 5/62

Keyboard input 5/54

Keypress detection 5/53, 5/61

Keypress trapping 5/62

KILL 4/10, 5/64

## L

Leaving GW-BASIC 1/3

LEFT\$ 5/65

LEN 5/65

LET 5/66

LINE 5/67

Line

- drawing 5/67

- format 1/9

- length 1/9

- number 1/9

- style 5/67

LINE INPUT 3/6, 5/69

LINE INPUT # 4/11, 5/69

Lines

- adding new 3/2

- deleting 3/2

- logical 3/2

- modifying 3/2

LIST 1/9, 3/3, 5/70

Listing a program 5/70

Listing to a device 5/70

Listing to a disk 5/70

Listing to a printer 5/70, 5/72

Literals - string 6/6

LLIST 5/72

LOAD 4/9, 5/72

Loading a program 5/72

LOC 4/11, 5/73

LOCATE 5/74

LOCK 5/75

LOF 4/11 5/77

Logarithms 5/37, 5/77, 5/77

Logical line definition 3/6

Logical lines 3/2

Logical operators 2/16, 2/21

Loudspeaker sound 5/143

LPOS 5/78

LPRINT 5/78

LPRINT USING 5/78

LSET 5/79, 5/124

LST 4/7

## M

Mathematical functions

not GW-BASIC intrinsic 7/13

Memory 1/6, 5/4, 5/5, 5/13, 5/25, 5/46, 5/79, 5/105, 6/1

out of 3/3

variables in 5/157, 5/158

MERGE 4/10, 5/7, 5/80

MID\$ 5/80, 5/81

MKD\$ 4/16, 5/82

MKDIR 4/16, 5/82

MKI\$ 4/16, 5/82

MKS\$ 4/16, 5/82

Modes of operation 1/8

Modifying lines 3/2

Modulus arithmetic 2/12, 2/14

Monochrome display 5/137

MS-DOS

commands and programs 5/142

environment 5/30

Multiplication 2/12

Music 5/91, 5/105

queue 5/108

## N

NAME 4/10, 5/83

Negation 2/12

Nested loops 5/44

NEW 5/84

NOT 2/12

Numbers

- arguments 6/9

- double precision 6/2, 6/8

- hexadecimal 2/2

- internal representation of 6/2

- single precision 6/2, 6/8

Numeric

- constants 2/6

- conversion into strings 4/16

  - double precision 2/8

  - single precision 2/8

- display format 5/114, 5/117

- values into strings 4/16

- variables 2/9

  - names

## O

OCT\$ 5/84

Octal 2/7, 5/84

ON COM 5/85

ON ERROR GOTO 5/86

ON GOSUB 2/25

ON KEY 2/24, 5/87

ON PLAY 2/24, 5/91

ON STRIG 2/24, 5/92

ON TIMER 2/24, 5/93

ON...GOSUB 5/87

OPEN 4/11

OPEN COM 5/95, 5/97

Operation modes 1/8

- Operators 2/12
  - arithmetic 2/13
  - logical 2/16, 2/21
  - order of precedence 2/12
  - relational 2/12, 2/14
  - string 2/19
- Option base 5/100
- OR 2/12
- OUT 5/100
- Out of memory 3/3
- Output re-direction 4/7
- Overflow 2/15

## P

- PALETTE 5/103
- PALETTE USING 5/103
- PAINT, 5/125
- Parent directory 4/5
- Passing statements 3/2
- Pathname 4/4, 5/11
  - length 4/4
- Paths 4/4
- PCOPY 5/104
- PEEK 5/105
- Pixel 5/67, 5/112
- Pixel color 5/110, 5/123
- PLAY 5/105, 5/108
- PLAY OFF 5/108
- PLAY ON 5/108
- PLAY STOP 5/108
- PMAP 5/109
- Point plotting 5/110, 5/112, 5/123
- POKE 5/111
- Port 5/16
  - input 5/54, 5/161
  - output 5/100
- POS 5/20, 5/112
- PRESET 5/112
- Print formatting 5/151
- Print head position 5/78, 5/113
- PRINT USING 5/116
- PRINT USING # 4/11, 5/121

PRINT # 4/11, 5/121, 5/124  
Printer line width 5/163  
Printing 5/78  
Program  
    annotation 5/129  
    chaining 5/7  
    continuation 5/18  
    data 5/21, 5/128, 5/132  
    debugging 1/8, 5/153  
    deletion 5/7, 5/84  
    editing 3/3  
        delete characters 3/3  
        insert characters 3/3  
        type over characters 3/3  
    file commands 4/9  
    files protecting 4/10  
    line deletion 5/26  
    line renumbering 5/7, 5/130  
    loops 5/162  
    merging 5/7, 5/80  
    running 5/135  
    saving 5/136  
    suspension 5/161  
    termination 5/30, 5/146  
    tracing 5/153, 3/2  
Prompt string 5/54, 5/69  
Protecting program files 4/10  
PSET 5/123  
PUT (graphics) 5/125  
PUT (i/o) 5/124

## R

Random access file 4/14, 5/32, 5/47, 5/73, 5/79

- accessing 4/17

- buffer 4/16, 5/39

- creating 4/15

- operations 4/19

- record length 4/16

- writing 5/124

Random number 5/126, 5/135

RANDOMISE 5/126

Re-direction of standard I/O 4/7

Reading memory contents 5/105, 5/128

Reading sequential files 4/13

Record

- locking 5/75

- number 5/73

- protection 5/154

- unlocking 5/75

'Redo from start' message 5/55

Referencing arguments 6/4

Registers - DX 6/10

Relational operators 2/12, 2/14

Reserved words 7/11

RESET 5/132

RESTORE 5/132

RESUME 5/133

RETURN 2/26, 5/133

RIGHT\$ 5/134

RMDIR 4/6, 5/134

RND 5/135

Root directory 4/5

RSET 5/79, 5/124

Rules for coding subroutines 6/5

RUN 1/8, 3/2, 4/9, 5/135

## S

SAVE 4/9, 4/10, 5/136

Screen

- attributes and color 5/140

- character -reading 5/141

- clear 5/15, 5/137, 5/141

- coordinates 5/159, 5/164

- copying 5/104

- duplication 5/104

- editor 3/2

- formatting 5/116

- mode specification 5/139

- text window 5/161

Segment address 5/25

Sequential file 4/11, 5/32, 5/56, 5/69

- adding to 4/14

- creating 4/12

- reading 4/13

- writing 5/121, 5/168

Setting up arrays 5/27

SGN 5/142

Sharing variables 5/17

SHELL 5/142

Signalling errors 3/3

SIN 2/23, 5/143

Single precision 2/10, 4/16, 5/19, 5/24, 5/143, 6/2, 6/8

SOUND 5/143

SPACE\$ 5/144

SPC 5/145

Special characters 2/3

SQR 2/23, 5/145

Square root 5/145

Stack 5/13

Standard I/O re-direction 4/7

Standard input device 5/53, 5/54, 5/57

Starting GW-BASIC 1/3

Statements - passing 3/2

stdin 1/4

stdout 1/4

STICK 5/146

STOP 3/6, 5/18, 5/146

- Storing images 5/48
- STR\$ 5/147
- STRIG 5/148
- STRIG OFF 5/149
- STRIG ON 5/149
- STRIG STOP 5/149
- String
  - arguments 6/10
  - concatenation 2/19
  - constants 2/6
  - conversion 5/12, 5/20, 5/50, 5/82, 5/84, 5/147, 5/156
  - descriptor 6/6
  - length 5/65
  - literals 6/6
  - manipulation 5/58, 5/65, 5/80, 5/81, 5/134, 5/144, 5/149, 5/149
  - operators 2/19
  - variable names 2/9
  - variables 2/9, 5/24
- STRING\$ 5/149
- Subroutine
  - coding rules 6/5
  - assembly language 6/1
- Subtraction 2/12
- SWAP 5/150
- Switches - command line option 1/4
- Swapping variable contents 5/150
- Syntax errors 3/6
- SYSTEM 5/150

## T

- TAB 5/151
- TAN 5/151
- Tangent 5/151
- Telephone communications 5/47, 5/151
- Text - entering 3/2
- Tiling 5/102
- Time
  - events 5/93, 5/152, 5/152, 5/153
  - format 5/152
  - reading 5/152

TIME\$ 5/152  
TIMER OFF 5/153  
    ON 5/153  
    STOP 5/153  
Trigonometry 5/18, 5/143, 5/151  
TROFF 5/153  
TRON 5/153

## U

Unconditional branching 5/49  
UNLOCK 5/75, 5/154  
User-defined functions 2/23  
USR 5/155, 6/1, 6/8

## V

Value - sign of 5/142, 5/156  
Values  
    double precision 4/16  
    single precision 4/16  
Variable 2/9  
    array 2/11  
    assignment 5/66  
    double precision 2/10, 2/21  
    integer 2/10  
    names 2/9  
        numeric 2/9, 2/10  
    resetting 5/13  
    single precision 2/10  
    string 2/9  
VARPTR 5/157  
VARPTR\$ 5/158  
VIEW 5/159  
VIEW PRINT 5/161  
Visual display pages 1/10

## W

WAIT 5/161  
WHILE...WEND 5/162  
WIDTH 4/11, 5/163  
Wild cards 5/42, 5/64

WINDOW 5/164

Working directory 4/5

WRITE # 4/11, 5/124, 5/167, 5/168

Writing programs 3/2

    GW-BASIC editor 3/1

Writing to memory 5/111

## **X**

XOR 2/12







*GWBASIC*  
*supplement No. 1*

**Software Release:**

Interpreter	VR3.20.2
Compiler	VR2.02.3

Hercules is a registered trademark of Hercules Computer Technology Inc.

Information contained in this document is subject to change without notice and does not represent a commitment on the part of Apricot Computers plc. The software described in this manual is furnished under a license agreement. The software may be used or copied only in accordance with the terms of this agreement.

All rights reserved; no use or disclosure without written consent.

Copyright © Apricot Computers plc 1986

Published by

Apricot Computers plc  
111 Hagley Road  
Edgbaston  
Birmingham B16 8LB  
England

Part No. 11867271

**Purpose:**

The GWBASIC Interpreter and Compiler are enhanced in this release to provide HBASIC compatible support for the Apricot Hercules compatible Monochrome Graphics circuitry.

**Enhancements**

The SCREEN statement is changed to provide a new functionality for Mode 2. Originally, this mode supported CGA 640 x 200 graphics only.

All other modes are unchanged.

The Hercules compatible support is:

*SCREEN 0:* Monochrome text 80 x 25 cells, each cell of size 9 x 14 pixels.

*SCREEN 2:* Monochrome 720 x 348 resolution graphics.

In addition this release provides full support, as documented, for the sound and communications (serial and parallel). The following additions to the documentation are:

*PLAY:* Includes a V parameter to adjust the volume. The possible scale is 1 .. 15. The syntax is identical to the *tempo* parameters.

**Note:**

this option only works on a XEN-i Workstation

**IBM CGA Conversion**

The following steps are required to convert IBM CGA applications to Hercules Graphics:

- Change SCREEN 0 and 1 statements to SCREEN 2 (although SCREEN 0 is valid)
- Remove COLOR statements
- Change WIDTH 40 statements to WIDTH 80
- Change the aspect ratios to reflect the new resolution
- Change the x/y coordinates to compensate for the higher resolution monochrome screen

## **GWBASIC Command**

For compatibility with HBASIC, a command line switch is included (/h) to force the interpreter to default to the graphics mode. This ensures that programs that do not specifically request graphics mode will still run.

The switch is only valid with Hercules compatible video circuitry.

The command line syntax is:

GWBASIC [/h] [programe]

## **Technical Details**

The implementation is based upon a new int 10h interrupt handler which is installed (daisy-chained) when the interpreter or compiled program is loaded. In Graphics Mode, all int 10h calls are intercepted. In Text Mode, the int 10h set mode call is intercepted to provide the correct text-graphics switch. The following calls are processed by the new handler – the remainder are passed to the BIOS:

- 0 Set mode
- 2 Set cursor position
- 3 Read cursor position
- 6 Scroll window up
- 7 Scroll window down
- 8 Read character and attribute
- 9 Write character and attribute
- A Write character only
- C Write pixel dot
- E Write character teletype
- F Get video mode

